



Informatik 09 - Objektorientierte Programmierung mit Java

aktualisiert: 10. Juni 2026

Stunde 1+2

Programmieren: Sprachen
Karol-Java Übersetzung
Übersicht Befehlsarten

Stunde 3+4

Semikolon und geschweifte Klammern
Strukturierung von Programmcode - Syntax
Strukturierung von Programmcode - Semantik:
Methoden
Übersetzungstraining: Methoden
Karol mit Java und Struktogramme

Stunde 5

Programmablauf grafisch darstellen: Struktogramme

Struktogramm-Training

Stunde 6+7+8

Taschenrechner

Methodenköpfe
Methoden im Klassendiagramm
Attribute im Klassendiagramm
Attribute im Klassendiagramm
Struktogramme und Klassendiagramme

Stunde 9

Übersicht: Variablen
Besondere Methoden: Getter und Setter
Spieler programmieren
Besondere Methoden: Konstruktoren
Besondere Methoden: Main/Hauptprogramm

Stunde 1+2

Stunde 3+4

Stunde 5

Stunde 6+7+8

Taschenrechner

Stunde 9

Speichern

Blöcke

Code

// Kommentar

▶ Start

Beenden

MarkeSetzen

MarkeLöschen

LinksDrehen 1

RechtsDrehen 1

Schritt 1

Hinlegen 1

Aufheben 1

wiederhole 10 mal

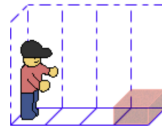
wiederhole solange

Start

Herzlich Willkommen! Hier lernst du Schritt für Schritt die Welt von Karol kennen. Das Tutorial zeigt dir die ersten Grundlagen für die Programmierung.

Tutorial anzeigen

Lege einen Ziegel



← zurück zur Übersicht

Struktogramm

Programmieren: Sprachen

Beim Programmieren schreibt man präzise und unmissverständliche Befehle, die dann vom **Compiler** in **Maschinsprache** übersetzt werden.

Hierfür gibt es verschiedene Programmiersprachen mit jeweils einer festgelegte Grammatik und Rechtschreibung (= **Syntax**). Dem gegenüber steht die **Logik eines Programms** (= **Semantik**), die weitgehend unabhängig von der Programmiersprache ist.

Bei den einsteigerfreundlichen block-basierten Sprachen wie Blockly oder Online-Karol übernehmen die Blöcke und ihre Verbindungsmöglichkeiten weitgehend die Einhaltung der Syntax. Bei text-basierten Sprachen wie **Java** oder **Python** müssen die Entwickler:innen selbst auf deren Einhaltung achten. Diese Sprachen bieten dadurch aber auch weitaus mehr Möglichkeiten (man sagt auch: sie sind mächtiger).

Dieses Skript arbeitet parallel mit den blockbasierten Sprachen Blockly und Online-Karol und der weit verbreiteten text-basierten und objektorientierten Sprache Java.

Karol-Java Übersetzung

1. Bearbeite die Aufgaben von Robot Karol online: karol.arrrg.de

2.

```
karol.schritt(2);  
karol.marksetzen();  
karol.schritt(10);  
while(true) {      karol.linksdrehen(1); }
```

```
if(karol.istZiegel()) {      karol.aufheben(1); }
```

```
if(karol.istOben()) {      karol.aufsteigen(1); } else {  
karol.linksdrehen(1); }
```

Übersicht Befehlsarten

Es gibt viele verschiedene Befehlsarten. Die für uns wichtigsten sind:

Art Online-Karol Blockly Java Struktogramm

Programmieren: Sprachen



Beim Programmieren schreibt man präzise und unmissverständliche Befehle, die dann vom Compiler in Maschinensprache übersetzt werden.

Hierfür gibt es verschiedene Programmiersprachen mit jeweils einer festgelegte **Grammatik und Rechtschreibung (= Syntax)**. Dem gegenüber steht die **Logik eines Programms (= Semantik)**, die weitgehend unabhängig von der Programmiersprache ist.

Bei den einsteigerfreundlichen **block-basierten Sprachen wie Blockly oder Online-Karol** übernehmen die Blöcke und ihre Verbindungsmöglichkeiten weitgehend die Einhaltung des Syntax. Bei **text-basierten Sprachen wie Python oder JavaScript** müssen die Entwickler:innen selbst auf deren Einhaltung achten. Diese Sprachen bieten dadurch aber auch weitaus mehr Möglichkeiten (**man sagt auch: sie sind mächtiger**).

Dieses Skript arbeitet parallel mit den blockbasierten Sprachen Blockly und Online-Karol und der weit verbreiteten text-basierten und **objektorientierten** Sprache **Java**.

Programmieren: Sprachen



Beim Programmieren schreibt man präzise und unmissverständliche Befehle, die dann vom **Compiler** in Maschinensprache übersetzt werden.

Hierfür gibt es verschiedene Programmiersprachen mit jeweils einer festgelegte **Grammatik und Rechtschreibung (= Syntax)**. Dem gegenüber steht die **Logik eines Programms (= Semantik)**, die weitgehend unabhängig von der Programmiersprache ist.

Bei den einsteigerfreundlichen **block-basierten Sprachen wie Blockly oder Online-Karol** übernehmen die Blöcke und ihre Verbindungsmöglichkeiten weitgehend die Einhaltung des Syntax. Bei **text-basierten Sprachen wie Python oder JavaScript** müssen die Entwickler:innen selbst auf deren Einhaltung achten. Diese Sprachen bieten dadurch aber auch weitaus mehr Möglichkeiten (**man sagt auch: sie sind mächtiger**).

Dieses Skript arbeitet parallel mit den blockbasierten Sprachen Blockly und Online-Karol und der weit verbreiteten text-basierten und **objektorientierten** Sprache **Java**.

Programmieren: Sprachen



Beim Programmieren schreibt man präzise und unmissverständliche Befehle, die dann vom **Compiler** in Maschinensprache übersetzt werden.

Hierfür gibt es verschiedene Programmiersprachen mit jeweils einer festgelegte **Grammatik und Rechtschreibung (= Syntax)**. Dem gegenüber steht die **Logik eines Programms (=)**, die weitgehend unabhängig von der Programmiersprache ist.

Bei den einsteigerfreundlichen **block-basierten Sprachen wie Blockly oder Online-Karol** übernehmen die Blöcke und ihre Verbindungsmöglichkeiten weitgehend die Einhaltung des Syntax. Bei **text-basierten Sprachen wie oder** müssen die Entwickler:innen selbst auf deren Einhaltung achten. Diese Sprachen bieten dadurch aber auch weitaus mehr Möglichkeiten (**man sagt auch: sie sind mächtiger**).

Dieses Skript arbeitet parallel mit den blockbasierten Sprachen Blockly und Online-Karol und der weit verbreiteten text-basierten und **objektorientierten** Sprache **Java**.

Programmieren: Sprachen



Beim Programmieren schreibt man präzise und unmissverständliche Befehle, die dann vom **Compiler** in Maschinensprache übersetzt werden.

Hierfür gibt es verschiedene Programmiersprachen mit jeweils einer festgelegte **Grammatik und Rechtschreibung (= Syntax**). Dem gegenüber steht die **Logik eines Programms (= Semantik**), die weitgehend unabhängig von der Programmiersprache ist.

Bei den einsteigerfreundlichen **block-basierten Sprachen wie Blockly oder Online-Karol** übernehmen die Blöcke und ihre Verbindungsmöglichkeiten weitgehend die Einhaltung des Syntax. Bei **text-basierten Sprachen wie oder** müssen die Entwickler:innen selbst auf deren Einhaltung achten. Diese Sprachen bieten dadurch aber auch weitaus mehr Möglichkeiten (**man sagt auch: sie sind mächtiger**).

Dieses Skript arbeitet parallel mit den blockbasierten Sprachen Blockly und Online-Karol und der weit verbreiteten text-basierten und **objektorientierten** Sprache **Java**.

Programmieren: Sprachen



Beim Programmieren schreibt man präzise und unmissverständliche Befehle, die dann vom **Compiler** in Maschinensprache übersetzt werden.

Hierfür gibt es verschiedene Programmiersprachen mit jeweils einer festgelegte **Grammatik und Rechtschreibung (= Syntax**). Dem gegenüber steht die **Logik eines Programms (= Semantik**), die weitgehend unabhängig von der Programmiersprache ist.

Bei den einsteigerfreundlichen **block-basierten Sprachen wie Blockly oder Online-Karol** übernehmen die Blöcke und ihre Verbindungsmöglichkeiten weitgehend die Einhaltung des Syntax. Bei **text-basierten Sprachen wie Java** **oder** müssen die Entwickler:innen selbst auf deren Einhaltung achten. Diese Sprachen bieten dadurch aber auch weitaus mehr Möglichkeiten (**man sagt auch: sie sind mächtiger**).

Dieses Skript arbeitet parallel mit den blockbasierten Sprachen Blockly und Online-Karol und der weit verbreiteten text-basierten und **objektorientierten** Sprache **Java**.

Programmieren: Sprachen



Beim Programmieren schreibt man präzise und unmissverständliche Befehle, die dann vom **Compiler** in Maschinensprache übersetzt werden.

Hierfür gibt es verschiedene Programmiersprachen mit jeweils einer festgelegte **Grammatik und Rechtschreibung (= Syntax**). Dem gegenüber steht die **Logik eines Programms (= Semantik**), die weitgehend unabhängig von der Programmiersprache ist.

Bei den einsteigerfreundlichen **block-basierten Sprachen wie Blockly oder Online-Karol** übernehmen die Blöcke und ihre Verbindungsmöglichkeiten weitgehend die Einhaltung des Syntax. Bei **text-basierten Sprachen wie Java oder Python** müssen die Entwickler:innen selbst auf deren Einhaltung achten. Diese Sprachen bieten dadurch aber auch weitaus mehr Möglichkeiten (**man sagt auch: sie sind mächtiger**).

Dieses Skript arbeitet parallel mit den blockbasierten Sprachen Blockly und Online-Karol und der weit verbreiteten text-basierten und **objektorientierten** Sprache **Java**.



1. Bearbeite die Aufgaben von Robot Karol online: karol.arrrg.de
2. Schalte in der Code-Ansicht die Sprache auf Java um und notiere auf dem AB jeweils zu welchem Java-Code der Block übersetzt wird. Achte hierbei auf Einrückungen!

Schritt 2

MarkeSetzen

Schritt 10

wiederhole immer

LinksDrehen 1

wiederhole 10 mal





1. Bearbeite die Aufgaben von Robot Karol online: karol.arrrg.de
2. Schalte in der Code-Ansicht die Sprache auf Java um und notiere auf dem AB jeweils zu welchem Java-Code der Block übersetzt wird. Achte hierbei auf Einrückungen!

Schritt 2

```
karol.schritt(2);
```

MarkeSetzen

Schritt 10

wiederhole immer

LinksDrehen 1

wiederhole 10 mal





1. Bearbeite die Aufgaben von Robot Karol online: karol.arrrg.de
2. Schalte in der Code-Ansicht die Sprache auf Java um und notiere auf dem AB jeweils zu welchem Java-Code der Block übersetzt wird. Achte hierbei auf Einrückungen!

Schritt 2

```
karol.schritt(2);
```

MarkeSetzen

```
karol.markeSetzen();
```

Schritt 10

wiederhole immer

LinksDrehen 1

wiederhole 10 mal





1. Bearbeite die Aufgaben von Robot Karol online: karol.arrrg.de
2. Schalte in der Code-Ansicht die Sprache auf Java um und notiere auf dem AB jeweils zu welchem Java-Code der Block übersetzt wird. Achte hierbei auf Einrückungen!

Schritt 2

```
karol.schritt(2);
```

MarkeSetzen

```
karol.markeSetzen();
```

Schritt 10

```
karol.schritt(10);
```

wiederhole immer

LinksDrehen 1

wiederhole 10 mal





1. Bearbeite die Aufgaben von Robot Karol online: karol.arrrg.de
2. Schalte in der Code-Ansicht die Sprache auf Java um und notiere auf dem AB jeweils zu welchem Java-Code der Block übersetzt wird. Achte hierbei auf Einrückungen!

Schritt 2

```
karol.schritt(2);
```

MarkeSetzen

```
karol.markeSetzen();
```

Schritt 10

```
karol.schritt(10);
```

wiederhole immer

LinksDrehen 1

```
while(true) {  
    karol.linksDrehen(1);  
}
```

wiederhole 10 mal





1. Bearbeite die Aufgaben von Robot Karol online: karol.arrrg.de
2. Schalte in der Code-Ansicht die Sprache auf Java um und notiere auf dem AB jeweils zu welchem Java-Code der Block übersetzt wird. Achte hierbei auf Einrückungen!

Schritt 2

```
karol.schritt(2);
```

MarkeSetzen

```
karol.markeSetzen();
```

Schritt 10

```
karol.schritt(10);
```

wiederhole immer

LinksDrehen 1

```
while(true) {  
    karol.linksDrehen(1);  
}
```

wiederhole 10 mal





1. Bearbeite die Aufgaben von Robot Karol online: karol.arrrg.de
2. Schalte in der Code-Ansicht die Sprache auf Java um und notiere auf dem AB jeweils zu welchem Java-Code der Block übersetzt wird. Achte hierbei auf Einrückungen!

Schritt 2

```
karol.schritt(2);
```

MarkeSetzen

```
karol.markeSetzen();
```

Schritt 10

```
karol.schritt(10);
```

wiederhole immer

LinksDrehen 1

```
while(true) {  
    karol.linksDrehen(1);  
}
```

wiederhole 10 mal





1. Bearbeite die Aufgaben von Robot Karol online: karol.arrrg.de
2. Schalte in der Code-Ansicht die Sprache auf Java um und notiere auf dem AB jeweils zu welchem Java-Code der Block übersetzt wird. Achte hierbei auf Einrückungen!

Schritt 2

```
karol.schritt(2);
```

MarkeSetzen

```
karol.markeSetzen();
```

Schritt 10

```
karol.schritt(10);
```

wiederhole immer

LinksDrehen 1

```
while(true) {  
    karol.linksDrehen(1);  
}
```

wiederhole 10 mal



Übersicht Befehlsarten



Es gibt viele verschiedene Befehlsarten. Die für uns wichtigsten sind:

Übersicht Befehlsarten



Es gibt viele verschiedene Befehlsarten. Die für uns wichtigsten sind:

Übersicht Befehlsarten



Es gibt viele verschiedene Befehlsarten. Die für uns wichtigsten sind:

Übersicht Befehlsarten



Es gibt viele verschiedene Befehlsarten. Die für uns wichtigsten sind:

Übersicht Befehlsarten



Es gibt viele verschiedene Befehlsarten. Die für uns wichtigsten sind:

Übersicht Befehlsarten



Es gibt viele verschiedene Befehlsarten. Die für uns wichtigsten sind:

Übersicht Befehlsarten



Es gibt viele verschiedene Befehlsarten. Die für uns wichtigsten sind:

Übersicht Befehlsarten



Es gibt viele verschiedene Befehlsarten. Die für uns wichtigsten sind:

Übersicht Befehlsarten



Es gibt viele verschiedene Befehlsarten. Die für uns wichtigsten sind:

Übersicht Befehlsarten



Es gibt viele verschiedene Befehlsarten. Die für uns wichtigsten sind:

Übersicht Befehlsarten



Es gibt viele verschiedene Befehlsarten. Die für uns wichtigsten sind:

Übersicht Befehlsarten



Es gibt viele verschiedene Befehlsarten. Die für uns wichtigsten sind:

Übersicht Befehlsarten



Es gibt viele verschiedene Befehlsarten. Die für uns wichtigsten sind:

Übersicht Befehlsarten



Es gibt viele verschiedene Befehlsarten. Die für uns wichtigsten sind:

Übersicht Befehlsarten



Es gibt viele verschiedene Befehlsarten. Die für uns wichtigsten sind:

Übersicht Befehlsarten



Es gibt viele verschiedene Befehlsarten. Die für uns wichtigsten sind:

Übersicht Befehlsarten



Es gibt viele verschiedene Befehlsarten. Die für uns wichtigsten sind:

Übersicht Befehlsarten



Es gibt viele verschiedene Befehlsarten. Die für uns wichtigsten sind:

Übersicht Befehlsarten



Es gibt viele verschiedene Befehlsarten. Die für uns wichtigsten sind:

Übersicht Befehlsarten



Es gibt viele verschiedene Befehlsarten. Die für uns wichtigsten sind:

Übersicht Befehlsarten



Es gibt viele verschiedene Befehlsarten. Die für uns wichtigsten sind:

Übersicht Befehlsarten



Es gibt viele verschiedene Befehlsarten. Die für uns wichtigsten sind:

Übersicht Befehlsarten



Es gibt viele verschiedene Befehlsarten. Die für uns wichtigsten sind:

Übersicht Befehlsarten



Es gibt viele verschiedene Befehlsarten. Die für uns wichtigsten sind:

Übersicht Befehlsarten



Es gibt viele verschiedene Befehlsarten. Die für uns wichtigsten sind:

Übersicht Befehlsarten



Es gibt viele verschiedene Befehlsarten. Die für uns wichtigsten sind:

Übersicht Befehlsarten



Es gibt viele verschiedene Befehlsarten. Die für uns wichtigsten sind:

Übersicht Befehlsarten



Es gibt viele verschiedene Befehlsarten. Die für uns wichtigsten sind:

Stunde 1+2

Stunde 3+4

Stunde 5

Stunde 6+7+8

Taschenrechner

Stunde 9

Semikolon und geschweifte Klammern



Wann macht man in Java ein Semikolon und wann setzt man geschweifte Klammern ein?

1. Überlege zuerst allein (2 Minuten)
2. Besprecht eure Überlegungen zu zweit und probiert sie in Online-Karol zur bestätigen (4 Minuten)

Strukturierung von Programmcode - Syntax



In Java wird Programmcode vor allem mit

strukturiert:

markieren das Ende eines **einfachen**

Befehls oder Methodenaufrufs (normalerweise am Ende der Zeile).

markieren **Code-Abschnitte**,

die wiederholt (Schleifen) oder bedingt (Bedingung) ausgeführt werden. Außerdem markieren sie den

Methodenrumpf und Anfang und Ende einer Klasse. Je nachdem, was durch die Klammern markiert wird, spricht man synonym zu der Klammern auch von der Klasse, Methode, Schleife, etc.



Computer Science Student :



Strukturierung von Programmcode - Syntax

In Java wird Programmcode vor allem mit **Semikolons und geschweiften Klammern** strukturiert:

markieren das Ende eines **einfachen**

Befehls oder Methodenaufrufs (normalerweise am Ende der Zeile).

markieren **Code-Abschnitte**,

die wiederholt (Schleifen) oder bedingt (Bedingung) ausgeführt werden. Außerdem markieren sie den

Methodenrumpf und Anfang und Ende einer Klasse. Je nachdem, was durch die Klammern markiert wird, spricht man synonym zu **Code-Block** der Klammern auch von **Code-Block** der Klasse, Methode, Schleife, etc.



Computer Science Student :

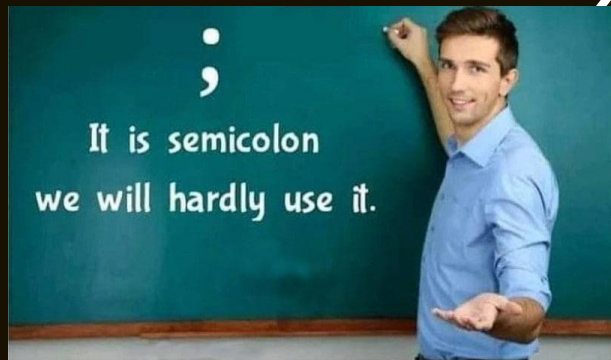


Strukturierung von Programmcode - Syntax

In Java wird Programmcode vor allem mit **Semikolons und geschweiften Klammern** strukturiert:

- **Semikolons** markieren das Ende eines **einfachen Befehls oder Methodenaufrufs** (normalerweise am Ende der Zeile).

Geschweiften Klammern markieren **Code-Abschnitte**, die wiederholt (Schleifen) oder bedingt (Bedingung) ausgeführt werden. Außerdem markieren sie den Methodenrumpf und Anfang und Ende einer Klasse. Je nachdem, was durch die Klammern markiert wird, spricht man synonym zu **Code-Abschnitten** der Klammern auch von **Code-Blöcken** der Klasse, Methode, Schleife, etc.



Computer Science Student :



Strukturierung von Programmcode - Syntax

In Java wird Programmcode vor allem mit **Semikolons und geschweiften Klammern** strukturiert:

- **Semikolons** markieren das Ende eines **einfachen Befehls oder Methodenaufrufs** (normalerweise am Ende der Zeile).
- **Geschweifte Klammern** markieren **Code-Abschnitte**, die wiederholt (Schleifen) oder bedingt (Bedingung) ausgeführt werden. Außerdem markieren sie den Methodenrumpf und Anfang und Ende einer Klasse. Je nachdem, was durch die Klammern markiert wird, spricht man synonym zu **Code-Abschnitt** der Klammern auch von **Code-Abschnitt** der Klasse, Methode, Schleife, etc.



Computer Science Student :



Strukturierung von Programmcode - Syntax

In Java wird Programmcode vor allem mit **Semikolons und geschweiften Klammern** strukturiert:

- **Semikolons** markieren das Ende eines **einfachen Befehls oder Methodenaufrufs** (normalerweise am Ende der Zeile).
- **Geschweifte Klammern** markieren **Code-Abschnitte**, die wiederholt (Schleifen) oder bedingt (Bedingung) ausgeführt werden. Außerdem markieren sie den Methodenrumpf und Anfang und Ende einer Klasse. Je nachdem, was durch die Klammern markiert wird, spricht man synonym zu **innerhalb** der Klammern auch von **außerhalb** der Klasse, Methode, Schleife, etc.



Computer Science Student :



Strukturierung von Programmcode - Syntax

In Java wird Programmcode vor allem mit **Semikolons und geschweiften Klammern** strukturiert:

- **Semikolons** markieren das Ende eines **einfachen Befehls oder Methodenaufrufs** (normalerweise am Ende der Zeile).
- **Geschweifte Klammern** markieren **Code-Abschnitte**, die wiederholt (Schleifen) oder bedingt (Bedingung) ausgeführt werden. Außerdem markieren sie den Methodenrumpf und Anfang und Ende einer Klasse. Je nachdem, was durch die Klammern markiert wird, spricht man synonym zu **innerhalb** der Klammern auch von **innerhalb** der Klasse, Methode, Schleife, etc.



Computer Science Student :



Strukturierung von Programmcode - Semantik: Methoden



Beim Programmieren können eigene Anweisungen/ Befehle/ Methoden/ Funktionen **deklariert** werden. Bei der **Deklaration beschreibt** zunächst mit dem `Methodenkopf`, welche Eigenschaften die Methode hat und anschließend im `Methodenrumpf`, wie genau sie sich verhalten soll. Eine deklarierte Methode kann anschließend beliebig oft verwendet (= **aufgerufen**) werden.

Hierdurch kann man sehr lange Programme in übersichtlichere Einzelteile zerlegen, sodass reduziert wird, der Code durch aussagekräftige `Methoden` besser verstanden werden kann und Fehler schneller gefunden werden.

In Java darf außerdem kein auszuführender Programmcode außerhalb von Methoden sein.

Die Deklaration besteht aus **Methodenkopf** und **Methodenrumpf** (von geschweiften Klammern umschlossen):

```
void tueEtwas() {  
  
    karol.schritt(2);  
}
```

Der **Aufruf** funktioniert wie ein einfacher Befehl:

```
tueEtwas();
```

Strukturierung von Programmcode - Semantik: Methoden



Beim Programmieren können eigene Anweisungen/ Befehle/ Methoden/ Funktionen **deklariert** werden. Bei der **Deklaration beschreibt** zunächst mit dem **Methodenkopf**, welche Eigenschaften die Methode hat und anschließend im **Methodenrumpf**, wie genau sie sich verhalten soll. Eine deklarierte Methode kann anschließend beliebig oft verwendet (= **aufgerufen**) werden.

Hierdurch kann man sehr lange Programme in übersichtlichere Einzelteile zerlegen, sodass reduziert wird, der Code durch aussagekräftige **Methoden** besser verstanden werden kann und Fehler schneller gefunden werden.

In Java darf außerdem kein auszuführender Programmcode außerhalb von Methoden sein.

Die Deklaration besteht aus **Methodenkopf** und **Methodenrumpf** (von geschweiften Klammern umschlossen):

```
void tueEtwas() {  
  
    karol.schritt(2);  
}
```

Der **Aufruf** funktioniert wie ein einfacher Befehl:

```
tueEtwas();
```

Strukturierung von Programmcode - Semantik: Methoden



Beim Programmieren können eigene Anweisungen/ Befehle/ Methoden/ Funktionen **deklariert** werden. Bei der **Deklaration beschreibt** zunächst mit dem **Methodenkopf**, welche Eigenschaften die Methode hat und anschließend im **Methodenrumpf**, wie genau sie sich verhalten soll. Eine deklarierte Methode kann anschließend beliebig oft verwendet (= **aufgerufen**) werden.

Hierdurch kann man sehr lange Programme in übersichtlichere Einzelteile zerlegen, sodass reduziert wird, der Code durch aussagekräftige besser verstanden werden kann und Fehler schneller gefunden werden.

In Java darf außerdem kein auszuführender Programmcode außerhalb von Methoden sein.

Die Deklaration besteht aus **Methodenkopf** und **Methodenrumpf** (von geschweiften Klammern umschlossen):

```
void tueEtwas() {  
  
    karol.schritt(2);  
}
```

Der **Aufruf** funktioniert wie ein einfacher Befehl:

```
tueEtwas();
```

Strukturierung von Programmcode - Semantik: Methoden



Beim Programmieren können eigene Anweisungen/ Befehle/ Methoden/ Funktionen **deklariert** werden. Bei der **Deklaration beschreibt** zunächst mit dem **Methodenkopf**, welche Eigenschaften die Methode hat und anschließend im **Methodenrumpf**, wie genau sie sich verhalten soll. Eine deklarierte Methode kann anschließend beliebig oft verwendet (= **aufgerufen**) werden.

Hierdurch kann man sehr lange Programme in übersichtlichere Einzelteile zerlegen, sodass **doppelter Code** reduziert wird, der Code durch aussagekräftige besser verstanden werden kann und Fehler schneller gefunden werden.

In Java darf außerdem kein auszuführender Programmcode außerhalb von Methoden sein.

Die Deklaration besteht aus **Methodenkopf** und **Methodenrumpf** (von geschweiften Klammern umschlossen):

```
void tueEtwas() {  
  
    karol.schritt(2);  
}
```

Der **Aufruf** funktioniert wie ein einfacher Befehl:

```
tueEtwas();
```

Strukturierung von Programmcode - Semantik: Methoden



Beim Programmieren können eigene Anweisungen/ Befehle/ Methoden/ Funktionen **deklariert** werden. Bei der **Deklaration beschreibt** zunächst mit dem **Methodenkopf**, welche Eigenschaften die Methode hat und anschließend im **Methodenrumpf**, wie genau sie sich verhalten soll. Eine deklarierte Methode kann anschließend beliebig oft verwendet (= **aufgerufen**) werden.

Hierdurch kann man sehr lange Programme in übersichtlichere Einzelteile zerlegen, sodass **doppelter Code** reduziert wird, der Code durch aussagekräftige **Methodennamen** besser verstanden werden kann und Fehler schneller gefunden werden.

In Java darf außerdem kein auszuführender Programmcode außerhalb von Methoden sein.

Die Deklaration besteht aus **Methodenkopf** und **Methodenrumpf** (von geschweiften Klammern umschlossen):

```
void tueEtwas() {  
  
    karol.schritt(2);  
}
```

Der **Aufruf** funktioniert wie ein einfacher Befehl:

```
tueEtwas();
```



Übersetze die folgenden Code-Schnipsel (**code snippets**) in Java-Code ohne den Computer zu verwenden.

Anweisung **tanzen**

- Schritt 1
- LinksDrehen 1
- Schritt 1
- LinksDrehen 2
- Schritt 1

Anweisung **ziegellmNordenLegen**

- wiederhole solange **NichtIst Norden**
- LinksDrehen 1
- wenn **NichtIstZiegel 1** dann
- Hinlegen 1

Anweisung **quadratLaufen**

- wiederhole immer
- Schritt 10
- LinksDrehen 1

- ziegellmNordenLegen
- quadratLaufen
- tanzen



Übersetze die folgenden Code-Schnipsel (**code snippets**) in Java-Code ohne den Computer zu verwenden.

Anweisung **tanzen**

- Schritt 1
- LinksDrehen 1
- Schritt 1
- LinksDrehen 2
- Schritt 1

Anweisung **ziegellmNordenLegen**

- wiederhole solange **NichtIst Norden**
 - LinksDrehen 1
- wenn **NichtIstZiegel 1** dann
 - Hinlegen 1

Anweisung **quadratLaufen**

- wiederhole immer
 - Schritt 10
 - LinksDrehen 1

- ziegellmNordenLegen**
- quadratLaufen**
- tanzen**

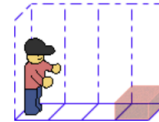
Karol mit Java und Struktogramme



- Bearbeite die Robot Karol (karol.arrrg.de) Level erneut. Schreibe den Programmcode dieses Mal direkt in Java!
Zur Erinnerung: Deinen Code schreibst du an der gelb markierten Stelle in der main-Methode.
- Zeige nach den Schritten jeweils das Struktogramm an und notiere auf einem **Schmierzettel**, wie du die letzte Spalte in Hefteintrag 6 befüllen würdest.
- Was passiert, wenn du dein Programm aus eigenen Methoden aufbaust? Wieso ist das so?

```
1 class Programm {  
2     Robot karol = new Robot();  
3  
4     void main() {  
5                               
6     }  
7 }
```

Lege einen Ziegel



← zurück zur Übersicht

Struktogramm

Stunde 1+2

Stunde 3+4

Stunde 5

Stunde 6+7+8

Taschenrechner

Stunde 9

Struktur und geschichtliche Entwicklung

Man stellt sich 6. Stück des Strukturs und setzt alle von geschichtliche Entwicklung auf

- Überlagerung von Methoden (2. Muster)
- Struktur von Methoden (3. Muster) und Struktur in der Klasse (4. Muster)

Strukturierung von Programmcode - Syntax

In Java wird Programmcode von oben mit **geschlossenen** und **geöffneten Klammern** strukturiert

- geschlossene Klammern** markieren das Ende eines Aufrufs
- geöffnete Klammern** markieren das Ende eines Aufrufs
- geschlossene Klammern** markieren Code-Blöcke, die vollständig (Schließen) oder fastig (Bedingte) ausgeführt werden. Aufrufe sind immer von dem Methodenaufruf und Anfang und Ende eines Aufrufs zu schließen, was durch die Klammern markiert wird, spricht man spricht in **geschlossenen** oder **geöffneten** Klammern, Methoden, Schichten, etc.

Strukturierung von Programmcode - Semantik, Methoden

Das Programmieren ist eine eigene Abstraktion der realen Welt. Die meisten Funktionen sind nicht nur die Methode, die Methode hat und produziert ein **Ergebniswert**, sie geben sie sich selbst mit. Die deklarative Methode kann unterschiedlich abstrahiert werden.

Methoden können eine oder einige Programme in unterschiedliche Einheiten zerlegen, sodass **Register** (z.B. Methodenaufruf) die Code durch verschiedene **Methoden** (z.B. Methodenaufruf) zerlegt werden können und falls in einer der Methoden eine **Methodenaufruf** (z.B. Methodenaufruf) verwendet wird, dann wird die Methode durch die Deklaration besteht aus **Methodenaufruf** und **Methodenaufruf** (z.B. Methodenaufruf).

```

class Test {
    public void main() {
        // ...
    }
}
    
```

Strukturierung von Programmcode - Semantik, Methoden

Methoden die Methodenaufruf (z.B. Methodenaufruf) in Java Code ohne über Computer zu verwenden.

Klass mit Java und Strukturierung

- Methoden die Methodenaufruf (z.B. Methodenaufruf) in Java Code ohne über Computer zu verwenden.
- Methoden die Methodenaufruf (z.B. Methodenaufruf) in Java Code ohne über Computer zu verwenden.
- Methoden die Methodenaufruf (z.B. Methodenaufruf) in Java Code ohne über Computer zu verwenden.
- Methoden die Methodenaufruf (z.B. Methodenaufruf) in Java Code ohne über Computer zu verwenden.

Programmablauf grafisch darstellen: Struktogramme

Struktogramme (auch: **Nassi-Shneiderman-Diagramme**) stellen Algorithmen (=Ablauf eines Programms) grafisch dar. Ein Struktogramm bezieht sich immer auf genau eine Methode und stellt aufgerufene Methoden als ein Element dar.

In Hefteintrag sind die Struktogramm-Bausteine für verschiedene Befehlsarten dargestellt, die beliebig miteinander kombiniert werden können.

Zusätzlich gibt es folgenden Baustein, der eine leere Stelle (z.B. leerer else-Block) darstellt:

Struktogramm-Training

Zeichne zu den folgenden Code Snippets jeweils ein Struktogramm:

Programmablauf grafisch darstellen: Struktogramme



Strukturgramme (auch: **Struktogramme**) stellen **Algorithmen (=Ablauf eines Programms)** grafisch dar. Ein Struktogramm bezieht sich immer auf genau eine Methode und stellt aufgerufene Methoden als ein Element dar.

In **Hefteintrag 6** sind die Struktogramm-Bausteine für verschiedene Befehlsarten dargestellt, die beliebig miteinander kombiniert werden können.

Zusätzlich gibt es folgenden Baustein, der eine leere Stelle (z.B. leerer else-Block) darstellt:

Programmablauf grafisch darstellen: Struktogramme



Strukturgramme (auch: **Nassi-Shneiderman-Diagramme**) stellen **Algorithmen (=Ablauf eines Programms)** grafisch dar. Ein Struktogramm bezieht sich immer auf genau eine Methode und stellt aufgerufene Methoden als ein Element dar.

In **Hefteintrag 6** sind die Struktogramm-Bausteine für verschiedene Befehlsarten dargestellt, die beliebig miteinander kombiniert werden können.

Zusätzlich gibt es folgenden Baustein, der eine leere Stelle (z.B. leerer else-Block) darstellt:

Zeichne zu den folgenden Code Snippets jeweils ein Struktogramm:



```
ziegelImNordenLegen();  
quadratLaufen();  
tanzen();
```

```
void quadratLaufen() {  
    while (true) {  
        karol.schritt(10);  
        karol.linksDrehen();  
    }  
}
```



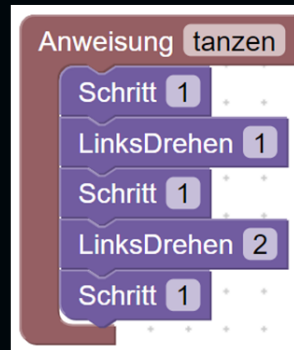


Zeichne zu den folgenden Code Snippets jeweils ein Struktogramm:



```
ziegelImNordenLegen();  
quadratLaufen();  
tanzen();
```

```
void quadratLaufen() {  
    while (true) {  
        karol.schritt(10);  
        karol.linksDrehen();  
    }  
}
```



Stunde 1+2

Stunde 3+4

Stunde 5

Stunde 6+7+8

Taschenrechner

Stunde 9

Stunde 1+2

Stunde 3+4

Stunde 5

Stunde 6+7+8

Taschenrechner

Stunde 9

Implementiere das Klassendiagramm rechts mit Hilfe von Blockly. Probiere deine Methode aus (siehe Anleitung auf der Vorderseite). Fülle die Lücken im Java-Code unten aus, wenn deine Methoden funktionieren.

Taschenrechner
+ double addieren(double z1, double z2)
+ double subtrahieren(double z1, double z2)
+ double multiplizieren(double z1, double z2)
+ double dividieren(double z1, double z2)
+ double mittelwert(double z1, double z2)

```
public class Taschenrechner
{
    public
```

```
}  
public
```

```
}  
public
```

```
}  
public
```

```
}
```

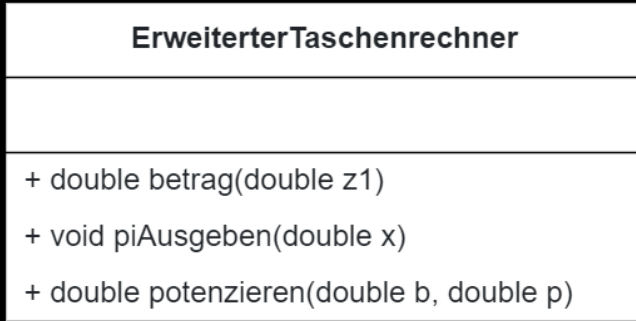
```
public
```

```
}
```

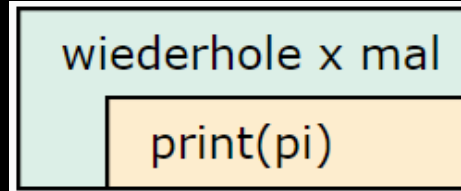
```
}
```


Setze folgende Diagramme in BlueJ-Blockly um und notiere den Programmcode auf der nächsten Seite

Klassendiagramm:



Struktogramm für `piAusgeben(double x)`:



Zeichne hier die Struktogramme für die restlichen Methoden:

```
public class ErweiterterTaschenrechner
{
    public

}
public
```


Programmablauf grafisch darstellen: Struktogramme

Struktogramme (auch: **Nassi-Shneiderman-Diagramme**) stellen Algorithmen (=Ablauf eines Programms) grafisch dar. Ein Struktogramm bezieht sich immer auf genau eine Methode und stellt aufgerufene Methoden als ein Element dar.

In Hefteintrag sind die Struktogramm-Bausteine für verschiedene Befehlsarten dargestellt, die beliebig miteinander kombiniert werden können.

Zusätzlich gibt es folgenden Baustein, der eine leere Stelle (z.B. leerer else-Block) darstellt:

Struktogramm-Training

Zeichne zu den folgenden Code Snippets jeweils ein Struktogramm:

Methodenköpfe

Im **Methodenkopf** können neben dem Namen noch weitere Informationen nach folgendem Schema gespeichert werden:

```
Rückgabewert Rückgabewerttyp name(Datentyp parametername) { // ... }
```

```
public void schritt1_aufen(int anzahl, ...) { // ... }
```

Methodenname und Parametername können weitgehend frei gewählt werden (Groß- und Kleinschreibung beachten!). Folgendes ist nicht:

- Leerzeichen im Namen, Zahlen am Anfang des Namens,
- Rechenzeichen im Namen, Schlüsselwörter als Name

Schlüsselwörter

```
public void schritt1_aufen(int anzahl, ...)
```

```
double Rückgabewert; String parametername; int anzahl; boolean return_value;
```

Methoden im Klassendiagramm

Attribute im Klassendiagramm

Findet heraus, wie Attribute im Klassendiagramm dargestellt werden. Achtet auf **Zugriffsmodifikator, Datentyp und Name**. Macht eine Skizze auf einem Schmierblatt.

Attribute im Klassendiagramm

Attribute werden im Klassendiagramm folgendermaßen dargestellt:

Struktogramme und Klassendiagramme



Im `Methodenkopf` können neben dem Namen noch weitere Informationen nach folgendem Schema gespeichert werden:

```
Zugriffsmodifizierer Rückgabety name(Datentyp parametername) { //...  
}  
public void schritteLaufen(int anzahl, ...) { //... }
```

Methodenname und Parametername können weitgehend frei gewählt werden (Groß- und Kleinschreibung beachten!). Folgendes ist **nicht** erlaubt:

Die verwendeten `Methodenköpfe` haben folgende Bedeutungen:

Methodenköpfe



Im **Methodenkopf** können neben dem Namen noch weitere Informationen nach folgendem Schema gespeichert werden:

```
Zugriffsmodifizierer Rückgabetyyp name(Datentyp parametername) { //...  
}  
  
public void schritteLaufen(int anzahl, ...) { //... }
```

Methodenname und Parametername können weitgehend frei gewählt werden (Groß- und Kleinschreibung beachten!). Folgendes ist **nicht** erlaubt:

Die verwendeten haben folgende
Bedeutungen:

Methodenköpfe



Im **Methodenkopf** können neben dem Namen noch weitere Informationen nach folgendem Schema gespeichert werden:

```
Zugriffsmodifizierer Rückgabety name(Datentyp parametername){ //...  
}  
public void schritteLaufen(int anzahl, ...){ //... }
```

Methodenname und Parametername können weitgehend frei gewählt werden (Groß- und Kleinschreibung beachten!). Folgendes ist **nicht** erlaubt: **Leerzeichen im Namen, Zahlen am Anfang des Namens, Rechenzeichen im Namen, Schlüsselwörter als Name**

Die verwendeten haben folgende
Bedeutungen:

Methodenköpfe

Im **Methodenkopf** können neben dem Namen noch weitere Informationen nach folgendem Schema gespeichert werden:

```
Zugriffsmodifizierer Rückgabety name(Datentyp parametername) { //...  
}  
public void schritteLaufen(int anzahl, ...) { //... }
```

Methodenname und Parametername können weitgehend frei gewählt werden (Groß- und Kleinschreibung beachten!). Folgendes ist **nicht** erlaubt: **Leerzeichen im Namen, Zahlen am Anfang des Namens, Rechenzeichen im Namen, Schlüsselwörter als Name**

Die verwendeten **Schlüsselwörter** haben folgende Bedeutungen:

Methodenköpfe

Im **Methodenkopf** können neben dem Namen noch weitere Informationen nach folgendem Schema gespeichert werden:

```
Zugriffsmodifizierer Rückgabety name(Datentyp parametername) { //...  
}  
public void schritteLaufen(int anzahl, ...) { //... }
```

Methodenname und Parametername können weitgehend frei gewählt werden (Groß- und Kleinschreibung beachten!). Folgendes ist **nicht** erlaubt: **Leerzeichen im Namen, Zahlen am Anfang des Namens, Rechenzeichen im Namen, Schlüsselwörter als Name**

Die verwendeten **Schlüsselwörter** haben folgende Bedeutungen:

Methodenköpfe

Im **Methodenkopf** können neben dem Namen noch weitere Informationen nach folgendem Schema gespeichert werden:

```
Zugriffsmodifizierer Rückgabety name(Datentyp parametername) { //...  
}  
public void schritteLaufen(int anzahl, ...) { //... }
```

Methodenname und Parametername können weitgehend frei gewählt werden (Groß- und Kleinschreibung beachten!). Folgendes ist **nicht** erlaubt: **Leerzeichen im Namen, Zahlen am Anfang des Namens, Rechenzeichen im Namen, Schlüsselwörter als Name**

Die verwendeten **Schlüsselwörter** haben folgende Bedeutungen:

Methodenköpfe

Im **Methodenkopf** können neben dem Namen noch weitere Informationen nach folgendem Schema gespeichert werden:

```
Zugriffsmodifizierer Rückgabety name(Datentyp parametername) { //...  
}  
public void schritteLaufen(int anzahl, ...) { //... }
```

Methodenname und Parametername können weitgehend frei gewählt werden (Groß- und Kleinschreibung beachten!). Folgendes ist **nicht** erlaubt: **Leerzeichen im Namen, Zahlen am Anfang des Namens, Rechenzeichen im Namen, Schlüsselwörter als Name**

Die verwendeten **Schlüsselwörter** haben folgende Bedeutungen:

Methodenköpfe

Im **Methodenkopf** können neben dem Namen noch weitere Informationen nach folgendem Schema gespeichert werden:

```
Zugriffsmodifizierer Rückgabety name(Datentyp parametername) { //...  
}  
public void schritteLaufen(int anzahl, ...) { //... }
```

Methodenname und Parametername können weitgehend frei gewählt werden (Groß- und Kleinschreibung beachten!). Folgendes ist **nicht** erlaubt: **Leerzeichen im Namen, Zahlen am Anfang des Namens, Rechenzeichen im Namen, Schlüsselwörter als Name**

Die verwendeten **Schlüsselwörter** haben folgende Bedeutungen:

Methodenköpfe

Im **Methodenkopf** können neben dem Namen noch weitere Informationen nach folgendem Schema gespeichert werden:

```
Zugriffsmodifizierer Rückgabety name(Datentyp parametername) { //...  
}  
public void schritteLaufen(int anzahl, ...) { //... }
```

Methodenname und Parametername können weitgehend frei gewählt werden (Groß- und Kleinschreibung beachten!). Folgendes ist **nicht** erlaubt: **Leerzeichen im Namen, Zahlen am Anfang des Namens, Rechenzeichen im Namen, Schlüsselwörter als Name**

Die verwendeten **Schlüsselwörter** haben folgende Bedeutungen:

Methoden im Klassendiagramm



Taschenrechner

- void einschalten()

+ double addieren(double z1, double z2)

```
public class Taschenrechner {  
    private void einschalten() {  
        // ...  
    }  
    public double addieren(double z1, double z2) {  
        return z1 + z2;  
    }  
}
```

Attribute im Klassendiagramm

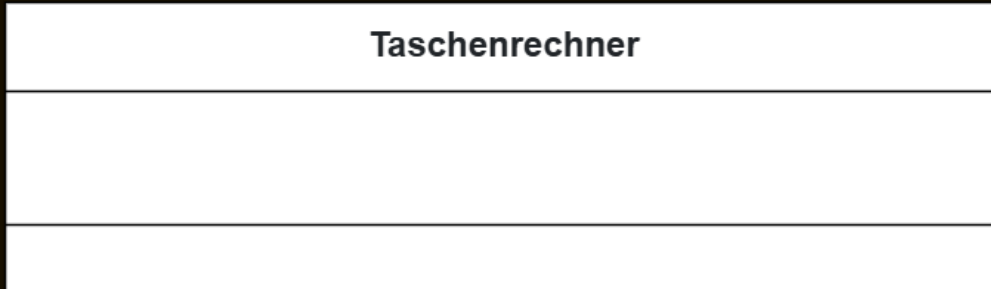


Findet heraus, wie Attribute im Klassendiagramm dargestellt werden. Achtet auf **Zugriffsmodifizierer, Datentyp und Name**. Macht eine Skizze auf einem Schmierblatt.

Attribute im Klassendiagramm



Attribute werden im Klassendiagramm folgendermaßen dargestellt:



Struktogramme und Klassendiagramme



Stunde 1+2

Stunde 3+4

Stunde 5

Stunde 6+7+8

Taschenrechner

Stunde 9

```
public class Spieler {  
    private double x;  
    private double y;
```

```
    public double getX() {
```

```
    }
```

```
    public double getY() {
```

```
    }
```

```
    public void act() {
```

```
}
```

```
public void                {
```

```
}
```

```
public void                {
```

```
}
```

```
public                    {
```

```
}
```

```
}
```

Methodenkopf

Im **Methodenkopf** können neben dem Namen noch weitere Informationen nach folgendem Schema gespeichert werden:

```

 Zugriffsmodifizierer Rückgabtyp Name (Datentyp Parametername) { (...) }
  
```

Wieder wird schematisch für `test.main()` ...

Methodenname und Parametername können weitgehend frei gewählt werden (Groß- und Kleinschreibung beachten). Folgendes ist nicht

Leertzeichen im Namen, Zahlen am Anfang des Namens, Rechenzeichen im Namen, Schlüsselwörter als Name

Schlüsselwörter

Zugriffsmodifizierer: public

```

double String int boolean
Rückgabtyp Rückgabtyp return-value
  
```

Methoden im Klassendiagramm

Attribute im Klassendiagramm

Findet heraus, wie Attribute im Klassendiagramm dargestellt werden. Achtet auf `Zugriffsmodifizierer, Datentyp und Name`. Macht eine Skizze auf einem Schmierblatt.

Attribute im UML-Klassendiagramm

Attribute werden im Klassendiagramm folgendermaßen dargestellt:

Klassendiagramme und Klassendiagramme

Initialisierte Variable

Initialisierung: ist die Zuweisung eines Werts nach der Deklaration. Schema: `name = Wert`. Beispiel: `alter = 14`.

Wertübergabe des Werts mit `*`, `++`, `--`, `*`, `+`, `-`, `name.equals(Wert)`

Beispiel: `int i = 0; int j = 1; int k = 2; int l = 3; int m = 4; int n = 5; int o = 6; int p = 7; int q = 8; int r = 9; int s = 10; int t = 11; int u = 12; int v = 13; int w = 14; int x = 15; int y = 16; int z = 17; int aa = 18; int bb = 19; int cc = 20; int dd = 21; int ee = 22; int ff = 23; int gg = 24; int hh = 25; int ii = 26; int jj = 27; int kk = 28; int ll = 29; int mm = 30; int nn = 31; int oo = 32; int pp = 33; int qq = 34; int rr = 35; int ss = 36; int tt = 37; int uu = 38; int vv = 39; int ww = 40; int xx = 41; int yy = 42; int zz = 43; int aa = 44; int bb = 45; int cc = 46; int dd = 47; int ee = 48; int ff = 49; int gg = 50; int hh = 51; int ii = 52; int jj = 53; int kk = 54; int ll = 55; int mm = 56; int nn = 57; int oo = 58; int pp = 59; int qq = 60; int rr = 61; int ss = 62; int tt = 63; int uu = 64; int vv = 65; int ww = 66; int xx = 67; int yy = 68; int zz = 69; int aa = 70; int bb = 71; int cc = 72; int dd = 73; int ee = 74; int ff = 75; int gg = 76; int hh = 77; int ii = 78; int jj = 79; int kk = 80; int ll = 81; int mm = 82; int nn = 83; int oo = 84; int pp = 85; int qq = 86; int rr = 87; int ss = 88; int tt = 89; int uu = 90; int vv = 91; int ww = 92; int xx = 93; int yy = 94; int zz = 95; int aa = 96; int bb = 97; int cc = 98; int dd = 99; int ee = 100;`

Objektreferenz → Erstellen einer Variable und Festlegung ihrer Eigenschaften.

lokale Variable → definiert mit `int` oder `String` in einem Code-Block und nur dort verfügbar.

Parameter → überträgt Werte von einer Methode zu einer anderen Methode.

return → überträgt den Wert einer Methode zurück zum Aufrufer.

main → die Methode, die das Programm startet.

Methoden: Getter und Setter

```

class Spieler {
    int id;
    String name;
    String position;

    // Getter
    int getId() { return id; }
    String getName() { return name; }
    String getPosition() { return position; }

    // Setter
    void setId(int id) { this.id = id; }
    void setName(String name) { this.name = name; }
    void setPosition(String position) { this.position = position; }
}
  
```

Spieler programmieren

```

class Spieler {
    int id;
    String name;
    String position;

    // Constructor
    Spieler(int id, String name, String position) {
        this.id = id;
        this.name = name;
        this.position = position;
    }

    // Getter
    int getId() { return id; }
    String getName() { return name; }
    String getPosition() { return position; }

    // Setter
    void setId(int id) { this.id = id; }
    void setName(String name) { this.name = name; }
    void setPosition(String position) { this.position = position; }
}
  
```

1. Programmieren die Klasse `Spieler` mit den Attributen `id`, `name` und `position`.
2. Schreiben einen Kontruktor und setzen dort die Werte für `id`, `name` und `position`.
3. Schreiben die Getter-Methoden `getId()`, `getName()` und `getPosition()`.
4. Schreiben die Setter-Methoden `setId()`, `setName()` und `setPosition()`.
5. Testen die Klasse `Spieler` mit dem `main`-Methode.

Besondere Methoden: Konstruktoren

```

class Spieler {
    // Konstruktoren
    Spieler() {
        // Default-Konstruktor
    }
    Spieler(int id, String name, String position) {
        // Parameter-Konstruktor
    }
}
  
```

Besondere Methoden: Main/Hauptprogramm

```

class Main {
    // Main-Methode
    public static void main(String[] args) {
        // Programm-Start
    }
}
  
```



Wertzuweisung mit Gleichheitszeichen **von rechts nach links**.

Schema: **name = Wert;**

Beispiel: **alter = 14;**

Verwendung des Werts mit Variablenname statt eines Werts

Beispiel 1: **System.out.println(alter);** statt **System.out.println(14);**

Beispiel 2: **alter = alter + 1;** statt **alter = 14 + 1;**

Vergleich des Werts mit **<, <=, ==, >=, >, !=, name.equals(Wert)**

Beispiele: **x == 5** (x = 5), **x != 5** (x ≠ 5), **text.equals("Hallo")** (text == "Hallo")





Wertzuweisung mit Gleichheitszeichen **von rechts nach links**.

Schema: **name = Wert;**

Beispiel: **alter = 14;**

Verwendung des Werts mit Variablenname statt eines Werts

Beispiel 1: **System.out.println(alter);** statt **System.out.println(14);**

Beispiel 2: **alter = alter + 1;** statt **alter = 14 + 1;**

Vergleich des Werts mit **<**, **<=**, **==**, **>=**, **>**, **!=**, **name.equals(Wert)**

Beispiele: **x == 5** (x **ist gleich** 5), **x != 5** (x **ist nicht gleich** 5), **text.equals("Hallo")** (text **ist gleich** "Hallo")





Wertzuweisung mit Gleichheitszeichen **von rechts nach links**.

Schema: **name = Wert;**

Beispiel: **alter = 14;**

Verwendung des Werts mit Variablenname statt eines Werts

Beispiel 1: **System.out.println(alter);** statt **System.out.println(14);**

Beispiel 2: **alter = alter + 1;** statt **alter = 14 + 1;**

Vergleich des Werts mit **<**, **<=**, **==**, **>=**, **>**, **!=**, **name.equals(Wert)**

Beispiele: **x == 5** (x **ist gleich** 5), **x != 5** (x **ungleich** 5), **text.equals("Hallo")** (text
)





Wertzuweisung mit Gleichheitszeichen **von rechts nach links**.

Schema: **name = Wert;**

Beispiel: **alter = 14;**

Verwendung des Werts mit Variablenname statt eines Werts

Beispiel 1: **System.out.println(alter);** statt **System.out.println(14);**

Beispiel 2: **alter = alter + 1;** statt **alter = 14 + 1;**

Vergleich des Werts mit **<, <=, ==, >=, >, !=, name.equals(Wert)**

Beispiele: **x == 5** (x **ist gleich** 5), **x != 5** (x **ungleich** 5), **text.equals("Hallo")** (text **ist "Hallo"**),





Wertzuweisung mit Gleichheitszeichen **von rechts nach links**.

Schema: **name = Wert;**

Beispiel: **alter = 14;**

Verwendung des Werts mit Variablenname statt eines Werts

Beispiel 1: **System.out.println(alter);** statt **System.out.println(14);**

Beispiel 2: **alter = alter + 1;** statt **alter = 14 + 1;**

Vergleich des Werts mit **<**, **<=**, **==**, **>=**, **>**, **!=**, **name.equals(Wert)**

Beispiele: **x == 5** (x **ist gleich** 5), **x != 5** (x **ungleich** 5), **text.equals("Hallo")** (text **ist "Hallo"**),





Wertzuweisung mit Gleichheitszeichen **von rechts nach links**.

Schema: **name = Wert;**

Beispiel: **alter = 14;**

Verwendung des Werts mit Variablenname statt eines Werts

Beispiel 1: **System.out.println(alter);** statt **System.out.println(14);**

Beispiel 2: **alter = alter + 1;** statt **alter = 14 + 1;**

Vergleich des Werts mit **<, <=, ==, >=, >, !=, name.equals(Wert)**

Beispiele: **x == 5** (x **ist gleich** 5), **x != 5** (x **ungleich** 5), **text.equals("Hallo")** (text **ist "Hallo"**),





Wertzuweisung mit Gleichheitszeichen **von rechts nach links**.

Schema: **name = Wert;**

Beispiel: **alter = 14;**

Verwendung des Werts mit Variablenname statt eines Werts

Beispiel 1: **System.out.println(alter);** statt **System.out.println(14);**

Beispiel 2: **alter = alter + 1;** statt **alter = 14 + 1;**

Vergleich des Werts mit **<, <=, ==, >=, >, !=, name.equals(Wert)**

Beispiele: **x == 5** (x **ist gleich** 5), **x != 5** (x **ungleich** 5), **text.equals("Hallo")** (text **ist "Hallo"**),



Getter haben den Zweck, den Wert eines (private) Attributs von anderen Klassen aus zugänglich zu machen. Der Methodenkopf folgt diesem Schema:

```
public TypAttribut getNameAttribut()
```

Setter ermöglichen das Setzen von Attributwerten von außerhalb derselben Klasse. Sie haben einen Parameter mit dem Datentyp des Attributs und Rückgabety `void`. Der Methodenkopf folgt diesem Schema:





Probieren nach jedem Schritt aus, was passiert, wenn du dein Spiel anstartest.

1. Programmiere die Klasse Spieler wie im Klassendiagramm vorgegeben. Programmiere `getX()` und `getY()` als Standard-Getter, lasse `act()` erstmal leer.
2. Erstelle einen Konstruktor und setze dort die Koordinaten auf feste Werte und finde damit heraus, wie groß das Spielfeld ist. Hierfür musst du etwas herumprobieren.
3. Notiere den Programmcode von `getX()` und `getY()` auf der nächsten Seite.
4. Das Spielfeld führt die Methode `act()` seiner Spieler-Objekte ca. 25x pro Sekunde aus. Finde heraus, wie du damit ein Spieler-Objekt auf dem Spielfeld bewegen kannst.
5. Wie kann man die Bewegungsgeschwindigkeit des Spielers festlegen? Probiere deine Idee im Programm aus und ergänze das Klassendiagramm.

Spieler

- double x
- double y

+ void act()
+ double getX()
+ double getY()

Besondere Methoden: Konstruktoren



Konstruktoren sind spezielle Methoden, die genutzt werden, um ein **Objekt zu initialisieren/konstruieren**. Ein Konstruktor wird **automatisch beim Erstellen eines Objekts** aufgerufen.

In Java erkennt man einen Konstruktor insb. an zwei Dingen:

1. Er **heißt**
2. Er hat **keinen**

Ansonsten verhält er sich wie eine Methode und kann z.B. Parameter haben, die dann oft als Startwerte für das Objekt dienen.

Ein typischer Konstruktor im Programmcode sieht z.B. so aus:

```
public Spieler(double startX , double startY) {  
x = startX;  
y = startY;  
}
```

Im Klassendiagramm würde dieses Beispiel so eingetragen:

Ein neues Objekt erstellt und speichert man dann z.B. so: **Spieler s1 = new Spieler(100,50);**

Besondere Methoden: Konstruktoren



Konstruktoren sind spezielle Methoden, die genutzt werden, um ein **Objekt zu initialisieren/konstruieren**. Ein Konstruktor wird **automatisch beim Erstellen eines Objekts** aufgerufen.

In Java erkennt man einen Konstruktor insb. an zwei Dingen:

1. Er **heißt genauso wie die Klasse**.
2. Er hat **keinen**.

Ansonsten verhält er sich wie eine Methode und kann z.B. Parameter haben, die dann oft als Startwerte für das Objekt dienen.

Ein typischer Konstruktor im Programmcode sieht z.B. so aus:

```
public Spieler(double startX , double startY) {  
x = startX;  
y = startY;  
}
```

Im Klassendiagramm würde dieses Beispiel so eingetragen:

Ein neues Objekt erstellt und speichert man dann z.B. so: **Spieler s1 = new Spieler(100,50);**

Besondere Methoden: Konstruktoren



Konstruktoren sind spezielle Methoden, die genutzt werden, um ein **Objekt zu initialisieren/konstruieren**. Ein Konstruktor wird **automatisch beim Erstellen eines Objekts** aufgerufen.

In Java erkennt man einen Konstruktor insb. an zwei Dingen:

1. Er **heißt genauso wie die Klasse** .
2. Er hat **keinen Rückgabotyp** .

Ansonsten verhält er sich wie eine Methode und kann z.B. Parameter haben, die dann oft als Startwerte für das Objekt dienen.

Ein typischer Konstruktor im Programmcode sieht z.B. so aus:

```
public Spieler(double startX , double startY) {  
x = startX;  
y = startY;  
}
```

Im Klassendiagramm würde dieses Beispiel so eingetragen:

Ein neues Objekt erstellt und speichert man dann z.B. so: **Spieler s1 = new Spieler(100,50);**

Besondere Methoden: Konstruktoren



Konstruktoren sind spezielle Methoden, die genutzt werden, um ein **Objekt zu initialisieren/konstruieren**. Ein Konstruktor wird **automatisch beim Erstellen eines Objekts** aufgerufen.

In Java erkennt man einen Konstruktor insb. an zwei Dingen:

1. Er **heißt genauso wie die Klasse** .
2. Er hat **keinen Rückgabotyp** .

Ansonsten verhält er sich wie eine Methode und kann z.B. Parameter haben, die dann oft als Startwerte für das Objekt dienen.

Ein typischer Konstruktor im Programmcode sieht z.B. so aus:

```
public Spieler(double startX , double startY) {  
x = startX;  
y = startY;  
}
```

Im Klassendiagramm würde dieses Beispiel so eingetragen:

+ Spieler(double startX, double startY)

Ein neues Objekt erstellt und speichert man dann z.B. so: **Spieler s1 = new Spieler(100,50);**

Besondere Methoden: Main/Hauptprogramm

Möchte man für ein Java-Programm starten, benötigt man eine Klasse. Diese nennt man auch den **Einstiegspunkt/Hauptprogramm** eines Programms. In ihr werden meistens die benötigten Klassen erstellt und ihre Methoden aufgerufen.

Der **Kopf** der Main-Methode ist immer gleich (siehe Bsp.):

```
public static void main() {  
    Spieler s1 = new Spieler(100,50);  
    // ... weitere Objekte und Methoden  
}
```

Das Schlüsselwort `public static` bedeutet hierbei, dass kein Objekt benötigt wird, um die Methode aufzurufen. In Blockly2Java wird die main-Methode automatisch zum Ausführen ausgewählt.

Besondere Methoden: Main/Hauptprogramm



Möchte man für ein Java-Programm starten, benötigt man eine **main-Methode**. Diese nennt man auch den **Einstiegspunkt/Hauptprogramm** eines Programms. In ihr werden meistens die benötigten Klassen erstellt und ihre Methoden aufgerufen.

Der **Kopf** der Main-Methode ist immer gleich (siehe Bsp.):

```
public static void main() {  
    Spieler s1 = new Spieler(100,50);  
    // ... weitere Objekte und Methoden  
}
```

Das Schlüsselwort **public static** bedeutet hierbei, dass kein Objekt benötigt wird, um die Methode aufzurufen. In Blockly2Java wird die main-Methode automatisch zum Ausführen ausgewählt.

Besondere Methoden: Main/Hauptprogramm



Möchte man für ein Java-Programm starten, benötigt man eine **main-Methode**. Diese nennt man auch den **Einstiegspunkt/Hauptprogramm** eines Programms. In ihr werden meistens die benötigten **Objekt** erstellt und ihre **Methoden** aufgerufen.

Der **Kopf** der Main-Methode ist immer gleich (siehe Bsp.):

```
public static void main() {  
    Spieler s1 = new Spieler(100,50);  
    // ... weitere Objekte und Methoden  
}
```

Das Schlüsselwort **public static void** bedeutet hierbei, dass kein Objekt benötigt wird, um die Methode aufzurufen. In Blockly2Java wird die main-Methode automatisch zum Ausführen ausgewählt.

Besondere Methoden: Main/Hauptprogramm



Möchte man für ein Java-Programm starten, benötigt man eine **main-Methode**. Diese nennt man auch den **Einstiegspunkt/Hauptprogramm** eines Programms. In ihr werden meistens die benötigten **Objekt** erstellt und ihre **Methoden** aufgerufen. Der **Kopf** der Main-Methode ist immer gleich (siehe Bsp.):

```
public static void main() {  
    Spieler s1 = new Spieler(100,50);  
    // ... weitere Objekte und Methoden  
}
```

Das Schlüsselwort `public static` bedeutet hierbei, dass kein Objekt benötigt wird, um die Methode aufzurufen. In Blockly2Java wird die main-Methode automatisch zum Ausführen ausgewählt.

Besondere Methoden: Main/Hauptprogramm



Möchte man für ein Java-Programm starten, benötigt man eine **main-Methode**. Diese nennt man auch den **Einstiegspunkt/Hauptprogramm** eines Programms. In ihr werden meistens die benötigten **Objekt** erstellt und ihre **Methoden** aufgerufen.

Der **Kopf** der Main-Methode ist immer gleich (siehe Bsp.):

```
public static void main() {  
    Spieler s1 = new Spieler(100,50);  
    // ... weitere Objekte und Methoden  
}
```

Das Schlüsselwort **static** bedeutet hierbei, dass kein Objekt benötigt wird, um die Methode aufzurufen. In Blockly2Java wird die main-Methode automatisch zum Ausführen ausgewählt.

