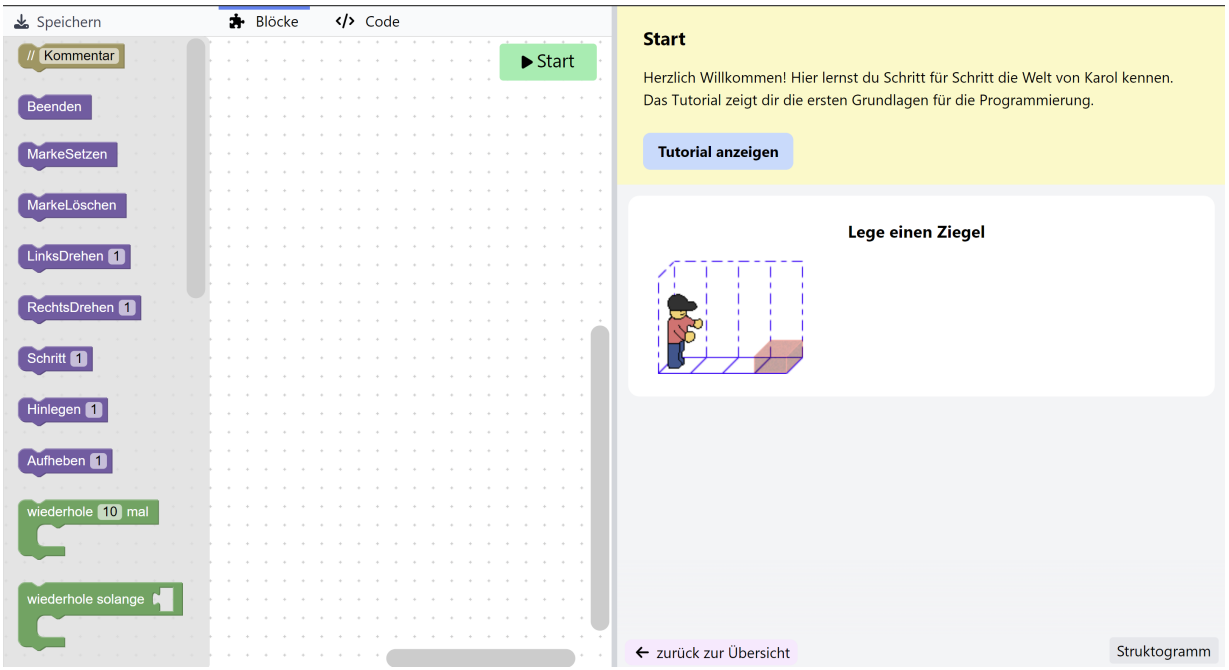
1 Programmieren: Sprachen

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**.



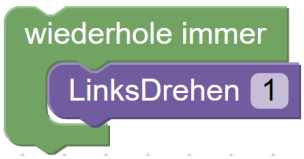
Karol-Java Übersetzung

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!

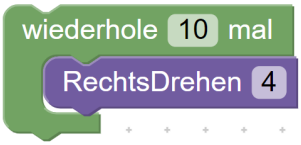

```
karol.schritt(2);
```


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


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

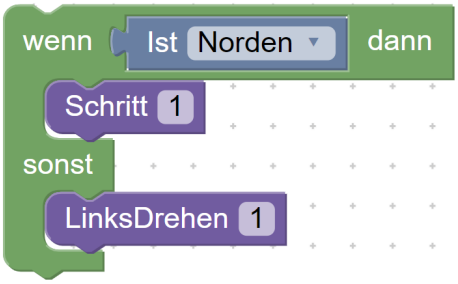



```
while(true) {
	karol.linksDrehen(1);
}
```




```
for(int i=0; i < 10; i++) {
	karol.rechtsDrehen(4);
}
```

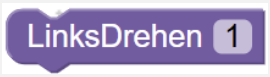

```
if(karol.istZiegel()) {
	karol.aufheben(1);
}
```



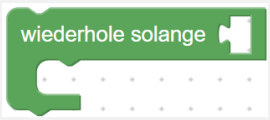

```
if(karol.istNordern()) {
	karol.aufheben(1);
} else {
	karol.linksDrehen(1);
}
```

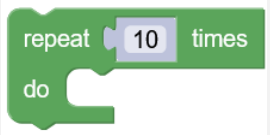
2 Übersicht Befehlsarten

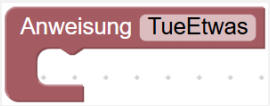
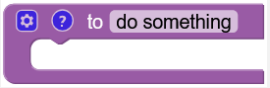
Es gibt viele verschiedene Befehlsarten. Die für uns wichtigsten sind:

Art	Online-Karol	Blockly	Java	Struktogramm
einf. Befehl			`karol.linksDre-hen(1);` `System.out.println(äbc");`	

Verzweigung			`if(<Bedingung>)`
			`{`
			`// wenn`
			`} else {`
			`// sonst`
			`}`	

Schleife / bedingte Wiederholung			`while(<Bedin-gung>) {`
			`...`
			`}`	

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

Methodendeklaration			`public void`
			`doSomething() {`
			`...`
			`}`	

Semikolon und geschweifte Klammern

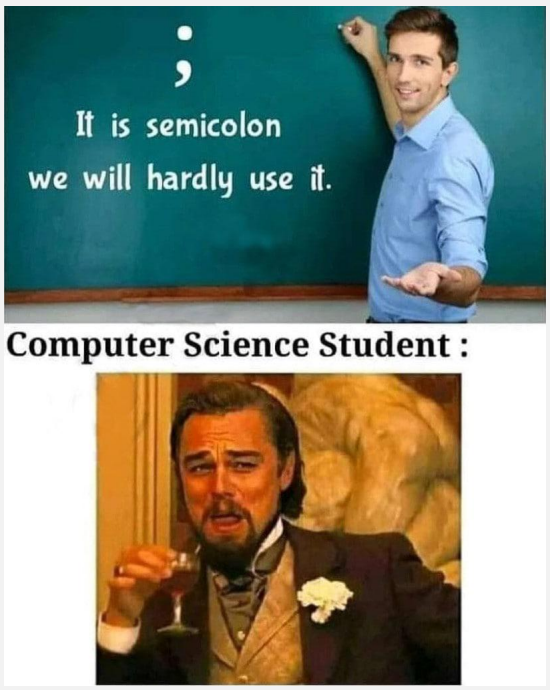
Wann macht man in Java ein Semikolon und wann setzt man geschweifte Klammern ein?

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

3 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.



www.reddit.com/r/ProgrammerHumor/comments/o0anud/semicolon

4 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):

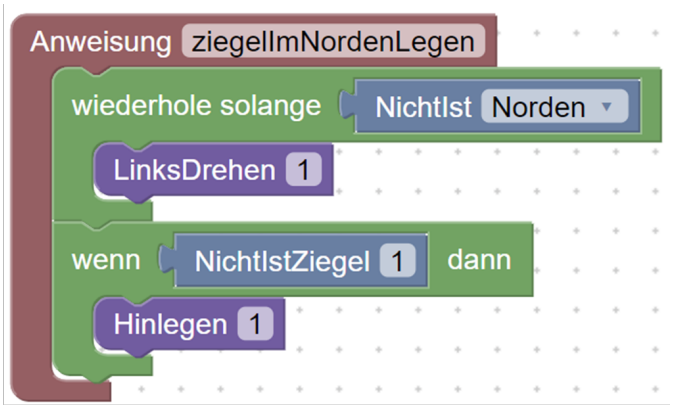
```
void tueEtwas() {
    karol.schritt(2);
}
```

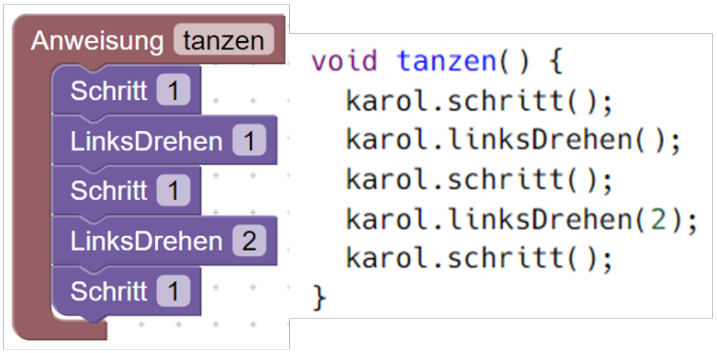
Der **Aufruf** funktioniert wie ein einfacher Befehl:

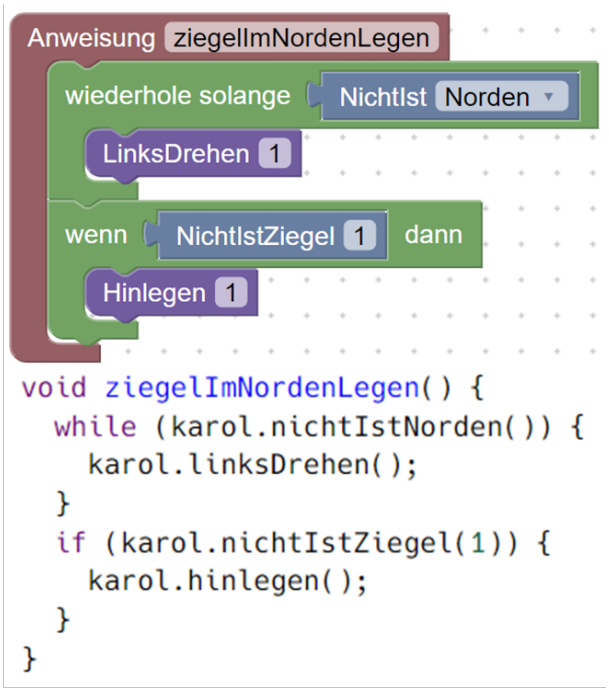
```
tueEtwas();
```

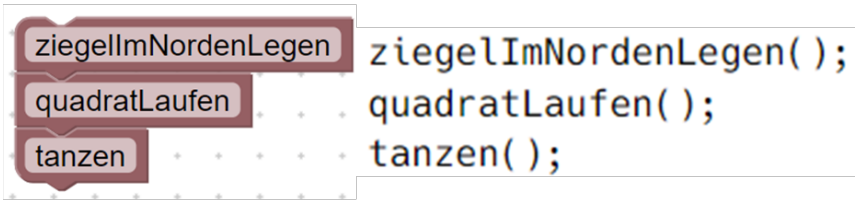
Übersetzungstraining: Methoden

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



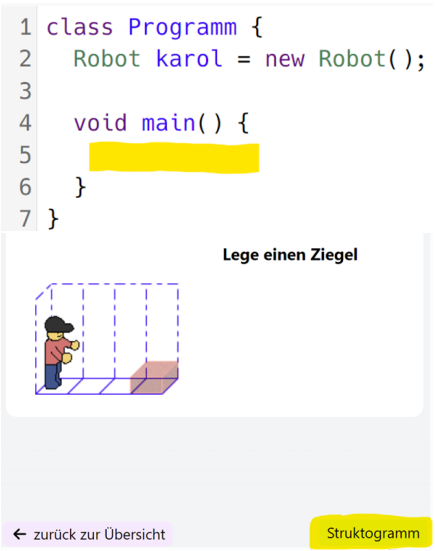






Karol mit Java und Struktogramme

- 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 2 befüllen würdest.
- Was passiert, wenn du dein Programm aus eigenen Methoden aufbaust? Wieso ist das so?



5 Programmablauf grafisch darstellen: Struktogramme

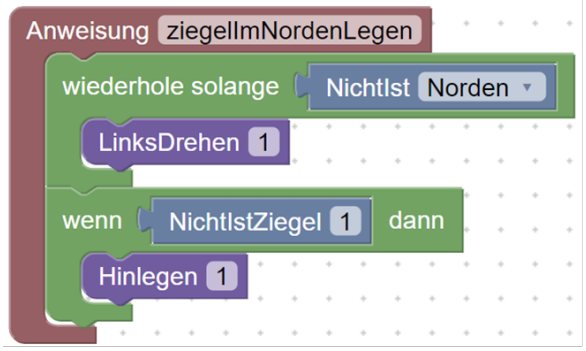
Struktorgramme (auch: `Nassi-Shneiderman-Diagramme`) stellen **Algorithmen (=Ablauf eines Programms)** grafisch dar. Ein Struktogramm bezieht sich immer auf genau eine Methode und stellt aufgerufene Methoden als ein Element dar.

In **Hefteintrag 2** 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:




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

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

6 Taschenrechner

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    double addieren(double z1, double z2) {


                return z1 + z2;
    }
```

```
    public    double subtrahieren(double z1, double z2)
      {

                return z1 - z2;
    }
    public    double multiplizieren(double z1, double
      z2) {

                return z1 * z2;
    }
    public    double dividieren(double z1, double z2) {


                return z1 + z2;
    }
    public    double mittelwert(double z1, double z2) {


                return (z1 + z2) / 2;
    }
}
```

7 Methodenköpfe

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

```
Zugriffsmodifizierer Rückgabetyp 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:

Zugriffsmodifizierer: public - Methode kann ‚überall' verwendet werden. (im Gegensatz dazu **private**: nur in der gleichen Klasse)

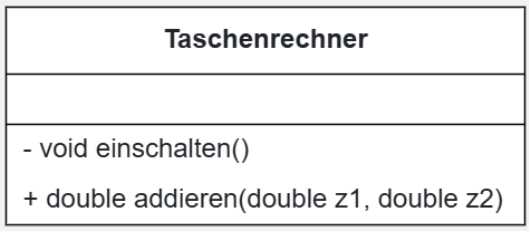
Datentyp - Art der Daten (z.B. Ganzzahlen: `int`, Kommazahlen: `double`, Text: `String`, Wahrheitswert: `boolean`)

Rückgabetyp - Datentyp des Rückgabewerts (= `return-value`). Kann z.B. Ergebnis einer Berechnung sein.

Besonderer Rückgabetyp - **void** bedeutet, die Methode hat keinen Rückgabewert. Außer bei void wird immer eine return-Anweisung benötigt.

Parameter - Speichert bei jedem Aufruf einen neuen Wert, der im Methodenrumpf über den Namen verwendet werden kann.

8 Methoden im Klassendiagramm



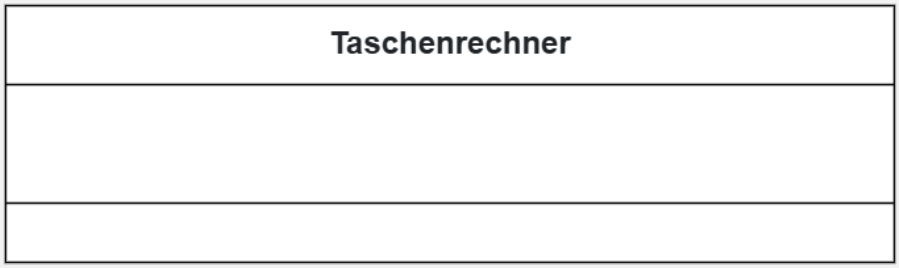

```
public class Taschenrechner {

    private void einschalten() {

        // ...

    }

    public double addieren(double z1,double z2) {

        return z1 + z2;

    }

}
```

Attribute im Klassendiagramm

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

9 Attribute im Klassendiagramm

Attribute werden im Klassendiagramm folgendermaßen dargestellt:



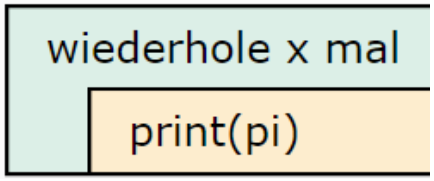
Struktogramme und Klassendiagramme

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

Klassendiagramm:

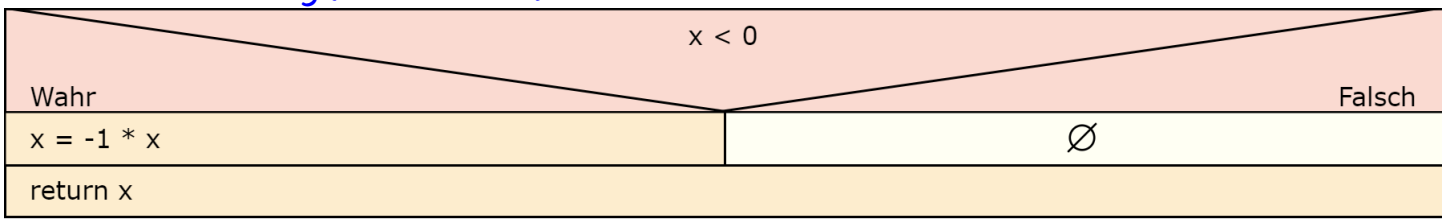
ErweiterterTaschenrechner
+ double betrag(double z1) + void piAusgeben(double x) + double potenzieren(double b, double p)

Struktogramm für piAusgeben(double x):

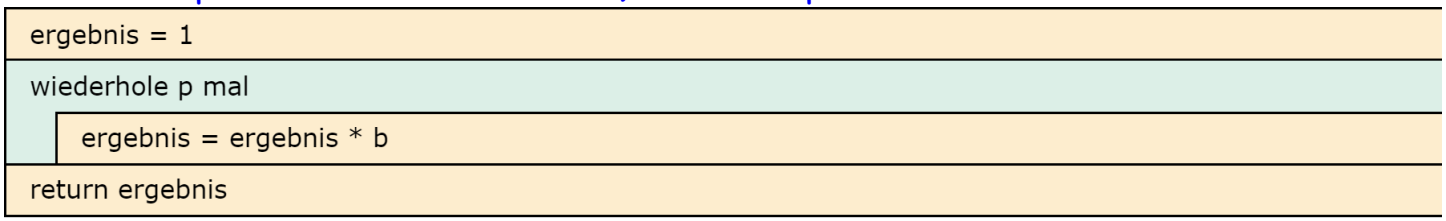


Zeichne hier die Struktogramme für die restlichen Methoden:

+ double betrag(double z1)



+ double potenzieren(double b, double p)



```
public class ErweiterterTaschenrechner
{
    public    double betrag(double z1) {

          if(z1 < 0) {

              z1 = -1 * z1;

          }

          return z1;
    }
    public    void piAusgeben(double x) {

          for(int i=0; i < x; i++) {

              System.out.println(Math.PI);

          }
    }
    public    double potenzieren(double b, double p) {

          double ergebnis = 1.0;

          for(int i=0; i < p; i++) {

              ergebnis = ergebnis * b;

          }

          return ergebnis;
    }
}
```

10 Übersicht: Variablen

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"`),

Deklaration = Erstellen einer Varibale und Festlegung ihrer Eigenschaften.

Arten von Variablen in der objektorientierten Programmierung:

- `lokale Variable` deklariert mit **Datentyp name;** in einem Code-Block und nur dort verfügbar.
- `Parameter` (manchmal auch: Input/Argument) deklariert mit **(Datentyp name)** in den Klammern des Methodenkopfs und nur in dieser Methode verfügbar. Wert wird bei jedem Aufruf neu festgelegt.
- `Attribut` deklariert am Anfang der Klasse mit **private Datentyp name;** Verfügbar in der ganzen Klasse mit anderem Wert für jedes Objekt. Zur Unterscheidung von lokalen Variablen kann bei der Verwendung auch **this.name** geschrieben werden.

11 Besondere Methoden: Getter und Setter

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

public TypAttribut getNameAttribut()

Spieler
- int x - int y - String imagePath

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

public void setNameAttribut(TypAttribut wert)

Standard-Getter/-Setter lesen bzw. schreiben den Wert des Attributs direkt (ohne Prüfung). Allgemein können Setter vor dem Schreiben den Parameter auf Gültigkeit prüfen oder modifizieren.

Spieler programmieren

Probiere nach jedem Schritt aus, was passiert, wenn du dein Spiel startest!

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 Spiel- ers festlegen? Probiere deine Idee im Programm aus und ergänze das Klassendiagramm und den Programmcode auf der nächsten Seite.
6. Wie kann man zu Beginn des Spiels die Startposition des Spielers (bei jedem neuen Objekt woanders) festlegen? Pro- biere auch das wieder aus und ergänze das Klassendia- gramm.
7. Über den Toolbox-Ordner 'Grafik: Objekte' kannst du in der Main weitere geometrische Formen hinzufügen. Nutze das, um ein kleines Spiel zu programmieren.

Spieler
- double x - double y
+ void act() + double getX() + double getY()

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

      private double speedX;

      private double speedY;

    public double getX() {

          return x;
    }
    public double getY() {

          return y;
    }
    public void act() {

          x += speedX;

          y += speedY;

          y += speedY;

          y += speedY;

          y += speedY;
    }

    public void   setX(double wert     {

          x = wert;
    }

    public void   setY(double wert     {

          y = wert;
    }

    public   Spieler(double startX, double startY     {

          x = wert;

          y = wert;
    }
}
```

12 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** `Rückgabetyp` .

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);**

13 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.