Thema: Greenfoot - Corona-Material 2 Datum: _____

Liebe Schülerinnen und Schüler des Jahrgangs EF,

wie ihr bestimmt mitverfolgt habt, ist leider noch nicht abschließend ersichtlich, wann wir uns im Kursverband wiedersehen können. Daher erhaltet ihr auf diesem Wege erneut einige Aufgaben, die ihr bitte bearbeitet. Bitte nicht erschrecken – Es sieht auf den ersten Blick nach mehr aus, als es letztlich ist. 😉 Bearbeitet bitte die Materialien M3 und M4 und beantwortet die jeweils dazugehörigen Arbeitsaufträge. Sie sind vom Arbeitsaufwand so konzipiert, dass ihr einen Materialblock pro Woche bearbeiten könnt. Aber natürlich dürft ihr auch direkt beide Materialblöcke bearbeiten. Das Material überbrückt also zunächst die Zeit bis zum 4. Mai.

Ich hoffe, euch geht es gut und ihr nutzt die Zeit produktiv, um hoffentlich bald und möglichst reibungslos zur Normalität und damit auch in den Schulalltag zurückzukehren.

Bleibt gesund und bis bald!

Liebe Grüße

D. Eckern

P.S.: Wie immer könnt ihr mich gerne bei Nachfragen über die Schul-E-Mail (d.eckern@ge-we.de) kontaktieren.

M3

Im Material M2 habt ihr bereits die For-Schleife als sogenannte „Zählschleife“ kennengelernt. Lest euch nachfolgend die Materialien M3.1 sowie M3.2 durch und bearbeitet den dazugehörigen Arbeitsauftrag A3.1!

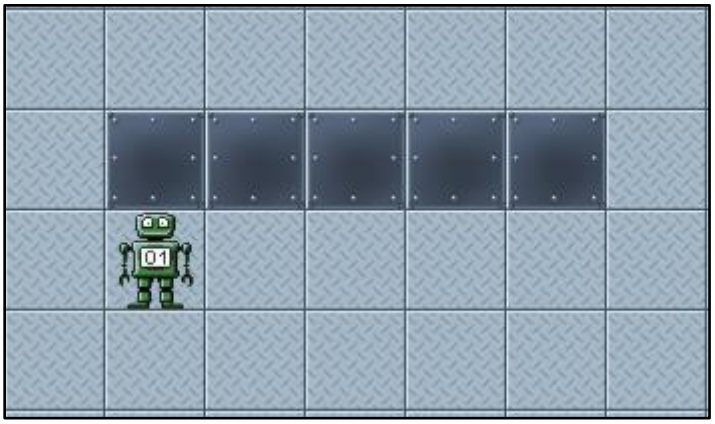


Abbildung M3.1 - Eine neue Ausgangssituation.

```
public void act()
{
    for(int i=0; i<4; i++)
    {
        if(wandLinks())
        {
            schraubeAblegen();
        }
        bewegen();
    }
```

Abbildung M3.2
For-Schleife zu dem Problem aus Abbildung M3.2.

Arbeitsaufträge:

A3.1) a) Beschreibe, was passiert, wenn die Methode act() (M3.2) ausführt wird.

b) Gebe an, wie oft die For-Schleife in M3.2 ausgeführt wird.

c) Es ist auch möglich, das gleiche Ergebnis mit einer while-Schleife zu erhalten. Formuliere eine while-Schleife, die das gleiche Ergebnis hervorbringt, wie die Ausführung der for-Schleife in Material M3.2. Ergänze dazu die nachfolgende Abbildung M3.3.

```
public void act()
{
                        // Zählen
    while(              )  // Bedingung einfügen
    {
                        //Robby legt die Schraube ab, falls... .

                        //Robby geht einen Schritt weiter und zählt
                        //dabei die Schritte hoch.

    }
}
```

A3.2)

a) Wie oft wird die nachfolgende Schleife durchlaufen? Begründe!

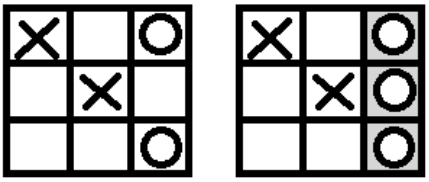
```
for (int i=1; i<10; i=1+2)
{
    System.out.println(i);          // Gibt Zahl i aus.
    if (i%2 ==0)                    // Prüft, ob i durch 2 (i:2) ohne Rest aufgeht.
    {
            System.out.prinln( „ Hat geklappt. “);          // Gibt Text aus.
    }
    else
    {
            System.out.println( „ Hat nicht geklappt. “);  // Gibt Text aus.
    }
}
```

b) Gebe an, was auf dem Bildschirm ausgegeben wird.

A3.3) Bei einem Tic-Tac-Toe-Spiel kann man, wenn man einem bestimmten System folgt, nicht verlieren.

a) Finden Sie dieses System heraus.

b) Entwickeln Sie einen umgangs-sprachlichen, aber eindeutigen Algorithmus, der genau dieses System wiedergibt.



Spielregeln:

- Ein Spieler zeichnet pro Zug ein Kreuz in das Gitter, der andere Spieler einen Kreis.
- Wer zuerst drei seiner Symbole in einer horizontalen, vertikalen oder diagonalen Reihe zeichnen konnte, gewinnt.

M4

Arbeitsaufträge zu M4:

A4.1) Lest euch den folgenden Text durch!

Ihr kennt inzwischen einige Bausteine, sogenannte *Methoden*, welche im Roboter-Programmiercode vorgegeben sind. Darunter sind unter anderem die *Methoden* bewegen() und schraubeAblegen() aufgelistet und ausformuliert. Sie sorgen dafür, dass jeder Roboter (egal ob Robby, Robita oder Robson) Schritte gehen und Schrauben ablegen kann.
Doch mit diesen (begrenzt vielen) *Methoden*, lassen sich nur einige Bewegungen programmieren. Um nach und nach mehr Bewegungen programmieren zu können, gibt es in Greenfoot natürlich auch die Möglichkeiten eigene *Methoden* zu erstellen. Diese können einem im Programmier-Alltag einige Arbeit ersparen.
Unabhängig von den Bewegungen des Roboters sind *Methoden* dazu da, die Verhaltensmöglichkeiten von Objekten (wie unseren Robotern) zu definieren. Sie geben also Aktionen und somit Fähigkeiten vor, mit denen ein Objekt der Klasse (hier die Klasse Roboter) in der Welt agieren kann.
Um jetzt das Programmieren zu vereinfachen, können wir eigene *Methoden* formulieren. Damit ein Roboter also eine Drehung um 180° vornehmen kann, musstet ihr bisher folgende *Methoden* in der act()-Methode des jeweiligen Roboters aufrufen:

z.B. im Code des Roboter Robby:

```
public void act()
{
        dreheRechts();
        dreheRechts();
}
```

Um die Drehung um die eigene Achse jedoch kürzer zu formulieren, kann eine neue Methode definiert werden, die diese Drehung um 180° direkt umsetzt.
Bisher wurde häufig nur die act()-Methode benutzt. Deren grundsätzlicher Aufbau ist jedoch gegenüber allen anderen in Prinzip gleich. Das Grundgerüst einer Methode wird durch ihre Signatur im *Methodenkopf* und die eigentliche Programmierung im *Methodenrumpf* festgelegt.

```
public void mName()                    // Methodenkopf
{
        Anweisungsblock                // Methodenrumpf
}
```

Die Signatur (mName) definiert bestimmte Eigenschaften der Methode, während im Quelltext des Anweisungsblocks die eigentliche Algorithmus-Erstellung passiert. Die Signalwörter *public* und *void* können momentan ohne Hintergrundwissen benutzt werden.

Statten wir unseren Roboter (z.B. Robby) nun mit einer weiteren Methode aus, so ist diese später unter den ausführbaren *Methoden* des Objekts zu erreichen und auszuführen:

```
public void dreheUm180()
{       dreheRechts();
        dreheRechts();
}
```

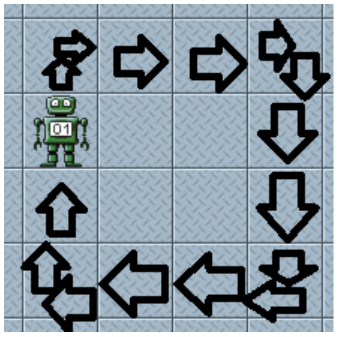
Methoden werden genutzt, um bestimmte zusammengehörige Programmteile zu strukturieren und unter einem eindeutigen Namen zusammenzufassen. Dabei sollte der *Methoden*name sinnvoll gewählt werden, allerdings auch nicht zu lang sein.

Ist diese Methode also einmal im **Roboter**-Programmcode eingefügt, kann ein jeder Roboter auf diese Methode zugreifen. Um dies dann zu tun, wird wieder die act()-Methode des Roboters (z.B. Robby) genutzt:

```
public void act()
{       dreheUm180();
}
```

A4.2)

a) Erstelle eine Methode, in der jeder Roboter folgenden Weg läuft:



- Hierbei darfst du den Namen der Methode frei wählen.
- Gebe den Programmcode an.
- In welchem Programmcode musst du diese Methode ergänzen?

A4.3) **Robby ist in einer ihm bisher unbekannten Welt und hat auch nur noch sehr wenig Energie. Daher will er sich jeweils nur sehr vorsichtig von einer Wand zur nächsten „schleichen“, um auf diese Weise nur wenig Energie zu verbrauchen. Robby kann jedoch nur dann weiter gehen, wenn hinter der Wand auch wirklich ein Akku liegt. Ansonsten ist das Risiko zu hoch, dass ihm unterwegs „der Saft ausgeht“.**

Nach Ablauf des Programms soll sich der Roboter hinter der letzten Wand befinden. Die Abstände zwischen den Wänden können beliebig sein, aber hinter jeder Wand außer der letzten liegt ein Akku.

Lösungshinweise:

Es lassen sich zwei Teilprobleme des Roboters zur Lösung dieses Szenarios finden.

a) Bis zur Wand laufen.
b) Um die Wand herumlaufen

a) Erkläre umgangssprachlich, wie eine Methode

- zurWand(), das Problem a) lösen kann.
- umDieWand(), das Problem b) lösen kann.
- wandSlalom(), das Gesamtproblem (a)+b)) mithilfe der beiden Methoden zurWand() und umDieWand() lösen kann.

b) Gebe ein PAP für die beiden Methoden zurWand() und umDieWand() an.

c) Entwickle für die Methoden zurWand() und umDieWand() einen passenden Programmcode.

d) Wie kann dann das Gesamtproblem in Robbys act()-Methode gelöst werden? Formuliere einen Programmcode, der das Gesamtproblem löst.