Dieses Beispiel baut auf den Kapiteln
Nested Objects, Prototyping, und Ereignissen auf!

Ant invasion

In diesem Beispiel werden **Objekte und Prototypen** thematisiert, indem eine Ameisenkolonie simuliert wird. Es eignet sich besonders gut als Teil einer Einführung in die objektorientierte Programmierung, beispielsweise am Anfang der 10. Jahrgangsstufe. Es wird wenig Vorwissen benötigt; dennoch wäre es gut, wenn die Zielgruppe bereits erste Erfahrungen mit blockbasierten Sprachen, z. B. in Form von Scratch gesammelt hat.
Wir verwenden neue Konzepte, die blockbasierte Sprachen uns ermöglichen - wie etwa **Tinkering** und **Direct Drive**. Explorativ werden hier Konzepte blockbasierter Programmierung wie **Nested Objects** und **Prototypen** erforscht.

Das Material besteht aus **Schülermaterialien** und **zusätzlichen Anmerkungen für Lehrkräfte** zur Umsetzung im Unterricht sowie **weiterführenden Erklärungen** und **Lösungen.**

Somit ist es möglich, das Material sowohl für die **eigene Weiterbildung** als auch in Auszügen für **Arbeitsblätter für den Schuleinsatz** zu verwenden.

Überblick über die verwendeten Symbole in diesem Handout:

	Aufgabe
	Bonusaufgabe für Fortgeschrittene, bzw. Benutzer mit Vorerfahrung
!	**Hinweis**
	Tinkering: hier gibt es keine falschen Antworten!
	Gespräch: Reflektion, Diskussion oder Austausch mit dem Banknachbarn
	Tipp: Hinweise, bzw. Ideen für die Umsetzung im Unterricht

Ant invasion

In diesem Modul simulieren wir das Verhalten einer Ameisenkolonie!

Vorgeschmack auf das fertige Projekt: bit.ly/snap-ants-solution

🔗 Link zum Projekt: bit.ly/snap-ants

Öffne das Projekt über den Link aus der Box.

Das Fach ist heute nicht nur **Informatik**, sondern auch **Biologie.** Für ein Referat ist es dein Job, zu erklären, **warum Ameisen sich in Ameisenstraßen bewegen.**

Du weißt natürlich bereits, dass das so ist, weil **Ameisen keine Gehirne haben - zumindest nicht so, wie wir Menschen.**

Stattdessen folgen sie **Pheromonen,** die **andere Ameisen hinterlassen.**

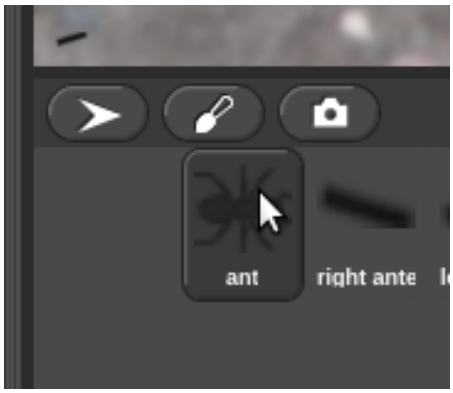
Das tut jede Ameise, und dadurch entsteht dann eine Ameisenstraße!

Am einfachsten wäre es natürlich, das Prinzip einfach anhand einer **Simulation** zu zeigen ...

...Und da du clever bist, hast du dich dazu entschlossen, die Simulation in Snap! zu programmieren!

(Keine Sorge, du musst kein Referat halten)

Nachdem wir das Projekt öffnen, sehen wir einige Objekte, sog. **Sprites**.



- Wähle das **ant-Objekt** aus.
- Uns interessiert momentan **nur** dieses Objekt - du musst in den anderen Objekten **noch nichts** tun!
- **Keine Objekte löschen:** Wenn Objekte gelöscht werden, musst du **wieder von vorne anfangen, da dieser Schritt NICHT rückgängig gemacht werden kann!**

Im **Skriptbereich** (große graue Fläche links) bauen wir Blöcke aus der **Blockpalette** zu Skripten zusammen und **führen sie aus, indem wir sie anklicken.**

Aufgabe:

Zunächst müssen wir **herausfinden**, **wie wir die Ameise laufen lassen können**. Baue daher **jeweils** ein Skript, mit dessen Hilfe die Ameise ...

1) … sich um ihre eigene Achse dreht!
2) … sich in einem Kreismuster auf der Leinwand bewegt!
3) … auf eine zufällige Position zeigt und sich 5 Schritte bewegt!
4) … auf eine zufällige Position zeigt, sich 5 Schritte bewegt, **und vom Rand abprallt!**
5) … auf eine zufällige Position zeigt, und sich 5 Schritte bewegt, **bis sie die Grenze der Leinwand berührt**!
6) … auf den **Mauszeiger zeigt** und sich auf ihn zubewegt, bis sie ihn berührt!

Tinkering:

- Schaffst du es, dass die Ameise eine Spur hinter sich her zieht?
- Schaffst du es, dass sie Muster auf der Leinwand malt?
- Schaffst du es, mit mehreren Ameisen zu malen? **Tipp:** 

Lösungen:

1)

2)

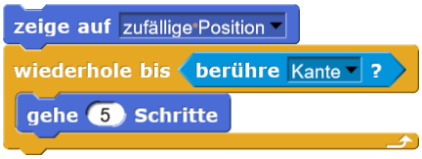
3)

/

/

4)

5)



6)

Anmerkung: Die Aufgaben sind bewusst unpräzise formuliert, da es hier (noch) **keine Probleme zu lösen gilt**. Ebenso sind die vorgegebenen Lösungen auch nicht als zwingend notwendig zu betrachten. Vielmehr soll erzielt werden, dass Schülerinnen und Schüler mit verschiedenen Blöcken und Blockkombinationen **experimentieren**, um zu verstehen, wie sie die Bewegung der Ameise auf der Leinwand steuern können.

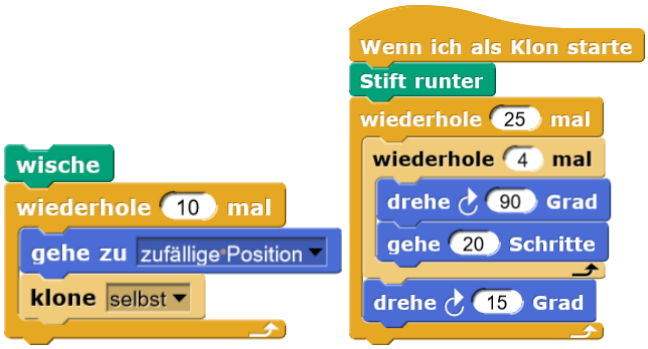
Mögliche Lösungen für die Tinkering-Phase:

- 

- 

/

/ …

- 

Tipps:

Es empfiehlt sich, den Schülerinnen und Schülern etwa 10-15 Minuten Bearbeitungszeit zu geben. Nach Ablauf der Bearbeitungszeit sollten die Ergebnisse **gesammelt und besprochen** werden, um sicherzustellen, dass die gesamte Gruppe mit demselben **Grundwissen weiterarbeitet.**

Obwohl das Grundprinzip dieses Moduls nicht auf Turtlegrafiken (bit.ly/w-turtle) basiert, eignen diese sich für einen **motivierenden Einstieg**, insbesondere in Verbindung mit Interaktivität (Steuerung durch den **Mauszeiger**).
Die Blöcke und Skripte, die die Schülerinnen und Schüler im Laufe dieser Aufgabe entwickeln, werden außerdem später aufgegriffen, um die Bewegungen der Ameisen zu implementieren.

Es ist nicht zu erwarten, dass die letzte Tinkeringaufgabe bereits “gelöst” wird. Sie dient primär als **Zeitpuffer**, und um eine erste Exploration mit dem Klon-Konzept zu ermöglichen.

Tipp: Es empfiehlt sich an dieser Stelle, zu erläutern, wo wir die im Lehrplan verwendeten Begriffe und Konzepte in Snap! finden. Das macht es einfacher für Schülerinnen und Schüler, das erlangte Wissen zu kontextualisieren und auf andere Programmiersprachen zu übertragen.

Methoden
Methoden entsprechen den Codeblöcken in Snap!. Sie können entweder global sichtbar oder auf ein einzelnes Objekt beschränkt sein. Genau wie Methoden können Codeblöcke **Eingabeparameter** (oder Inputs) haben, die sie **verarbeiten**, und eine **Ausgabe**, die sie zurückliefern. In Snap! ist diese Ausgabe oftmals **visueller** Natur, d.h. etwa, dass Objekte sich auf der Leinwand bewegen oder ihre Farbe ändern.

In der echten Welt folgen Ameisen einer **Pheromonspur,** die andere Ameisen für sie hinterlassen. Können wir eine solche Pheromonspur in Snap! nachbilden?

Wir blicken in die **Blockpalette von Snap!,** genauer gesagt in die Kategorie **Pen/Stift,** und finden dort u. A. folgende Blöcke:

Aufgabe: Erweitere das Skript der Ameise so, dass sie eine Pheromonspur hinter sich zieht! *(Tipp: Wähle eine gut sichtbare Stiftfarbe!)*

Klicke erneut auf das Skript, um es zu beenden. Starte es dann wieder. **Wird die alte Spur verworfen und eine neue Pheromonspur gemalt?**

!

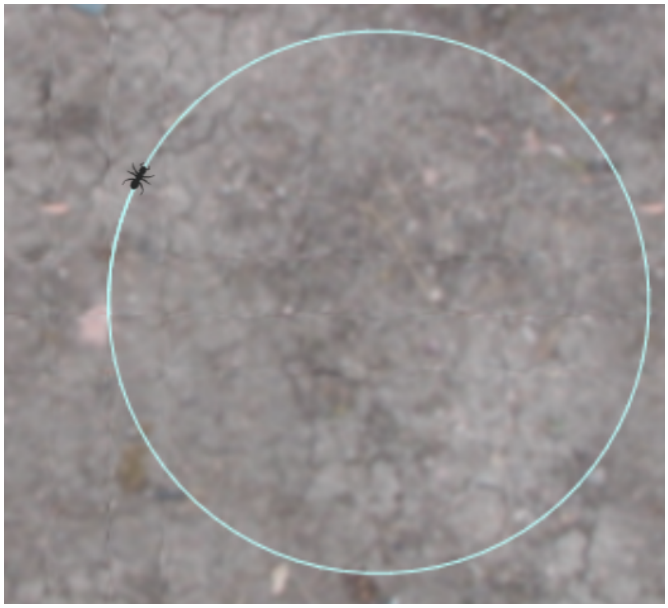
Die Spur “verläuft” sich mit der Zeit, d.h. sie wird **langsam unsichtbar.**
Das ist so gewollt - den Code dazu geben wir im Objekt **“trails”** vor.

IN DIESEM OBJEKT MUSST DU NICHTS TUN!

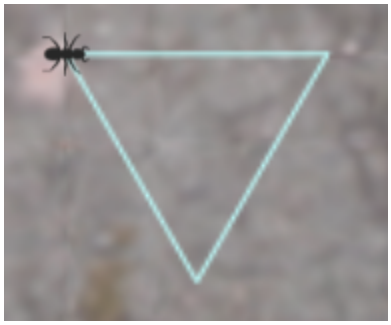
Tinkering:
Schaffst du es, die Ameise so laufen und malen zu lassen, dass sie folgende Muster auf der Leinwand malt?

a)

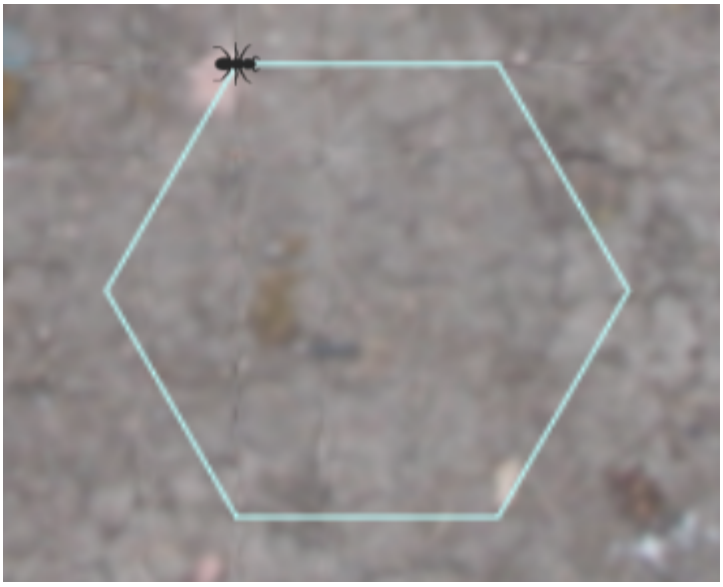
b)



c)



d)



Natürlich bewegen sich Ameisen im echten Leben **nicht so**.

Tipps:

Ob diese Aufgaben gelöst werden oder nicht ist für den weiteren Verlauf **nicht relevant.**
Diese **simplen und motivierenden** Aufgaben dienen dazu, die Schülerinnen und Schüler mit der Bewegung der Ameise experimentieren zu lassen, und **basieren auf Turtlegrafiken.**

Wir möchten das Skript nicht immer anklicken müssen. Stattdessen möchten wir es über den **grüne-Flagge-Knopf** starten.

Snap! ist eine **ereignisgesteuerte** Programmiersprache. Was bedeutet das?

- Blöcke, bzw. Skripte werden als Antwort auf ein **Ereignis** ausgeführt
- Ein solches Ereignis ist beispielsweise ein Klick auf das jeweilige Skript.

Die Blockkategorie Control/Steuerung zeigt uns weitere Ereignisse:

Block	Reagiert auf ...
	Klicken des grüne-Flagge-Knopfes
	Interaktion mit Tastatur
	Interaktion mit Mauszeiger
	Empfang von Nachrichten
	Erfüllung einer Bedingung
	Entstehung als Klon

Tritt ein Ereignis ein, so wird das jeweilige Skript, das am Block hängt, ausgeführt. Das können wir uns zunutze machen.

Aufgabe: Wir wollen das Skript nicht immer anklicken müssen, um es auszuführen! Dazu verwenden wir Ereignisse.
Manche Ereignisse sind hier sinnvoll, andere weniger.

Überlege dir, welche drei Ereignisse von oben sinnvoll sind. Du kannst sie einfach **ankreuzen (x)**.

Suche dir dann **ein** Ereignis aus und gib deinem Skript einen **Ereignis-Block!**

Tipp: An dieser Stelle bietet es sich im Unterricht an, die ereignisgesteuerte Programmierung in Snap! mit der **main**-Methode-Programmierung z. B. in Java zu vergleichen. Skripte werden ausgeführt, wenn Ereignisse eintreffen. Werden Skripte, bzw. Blöcke einfach angeklickt, um sie auszuführen, sprechen wir von **Direct Drive** - einem Spezialfall der ereignisgesteuerten Programmierung. Direct Drive basiert auf der **direkten Manipulation** von Programmen und ihrem Code und findet sich in vielen Programmiersprachen der blockbasierten Scratch-Sprachenfamilie.

Lösungen:

Sinnvolle Ereignisse für das Skript sind beispielsweise 

,


, oder 

.

Beispiel für das fertige Skript:

Tipp:

Auch 

kann verwendet werden. Hierbei handelt es sich um einen der beiden Bestandteile des **broadcasting-Konzepts**, das in Snap verwendet wird, um Nachrichten zwischen Objekten zu versenden.

>> Weiterführende Informationen und Aufgaben zu broadcasting: bit.ly/script-broad

Nun leben Ameisen aber natürlich nicht alleine, sondern in Kolonien.
Das wollen wir in unserer Simulation nachbilden.

Aufgabe: Beschreibe, wie du dieses Problem in einer anderen Programmiersprache angehen würdest!

_____

_____

_____

_____

Snap! ist eine klassenlose Programmiersprache. Was bedeutet das?

- In Snap! **gibt es keine** Klassen!
- Wir beschreiben konkrete Objekte, keine Klassen von Objekten
- Neue Objekte erzeugen wir demnach **nicht** aus Klassen, sondern als Kopien von bereits existierenden Objekten, sog. **Prototypen**.

Wie verwenden wir unsere Ameise nun als Prototyp für die Kolonie?
Zwei Blöcke spielen hierfür eine elementare Rolle:

	klont den angegebenen Prototypen (Auswahl durch Dropdown-Menü)
	Skript wird ausgeführt, wenn ein Klon erzeugt wird

Mithilfe dieser Blöcke können wir eine einfache Form des Klonens verwenden!

Aufgabe: Verändere deine Skripte so, dass **Prototyping** benutzt wird!

Der **Prototyp** soll folgendes tun:

- Stiftfarbe setzen
- Leinwand wischen
- Stift hochnehmen
- zu zufälliger Position gehen
- in zufällige Richtung zeigen
- Stift absetzen
- 15 Klone von sich selbst erzeugen

Die **Klone** sollen folgendes tun:

- In einer Endlosschleife:
- einige Schritte gehen
- vom Rand abprallen

Lösung:
In einer anderen Programmiersprache, beispielsweise Java, würden wir hierzu weitere Objekte der Klasse “Ameise” erzeugen.

Tipp: Am gewinnbringendsten ist diese Einheit, wenn Schülerinnen und Schüler vorher, bzw. nachher ein ähnliches Beispiel in einer textbasierten Sprache, wie etwa Java, implementieren. Dadurch wird es möglich, beide Herangehensweisen und Programmiersysteme miteinander zu vergleichen, und den Unterrichtsinhalt aus verschiedenen Perspektiven zu erfahren.

Tipp:
Alles was mit Klassen und Objekten implementierbar ist, ist auch mit Objekten und Prototypen implementierbar. **Die beiden Paradigmen sind gleich mächtig.**

>> Weiterführende Informationen und Aufgaben zu Prototyping: bit.ly/script-proto

An dieser Stelle wird nur eine **grundlegende** Form der prototypischen Vererbung implementiert. Einen detaillierteren Einblick in die Thematik in Snap! gibt es in der **separaten Einheit** dazu.

Tipp: Es empfiehlt sich an dieser Stelle, zu erläutern, wo wir die im Lehrplan verwendeten Begriffe und Konzepte in Snap! finden. Das macht es einfacher für Schülerinnen und Schüler, das erlangte Wissen zu kontextualisieren und auf andere Programmiersprachen zu übertragen.

Klassen
Snap! ist eine **klassenlose** Programmiersprache. Anstatt abstrakte Baupläne von Objekten zu beschreiben, arbeitet der Benutzer stattdessen mit konkreten Objekten. Möchten wir dennoch auf konzeptioneller Ebene Klassen thematisieren, so können wir das auch in Snap! tun:

- Jedes **Objekt** ist ein **Objekt der Klasse “Sprite”**
- Daher starten alle neuen Objekte mit dem Standardkostüm (Raumschiff) und allen global sichtbaren Blöcken im Programm, es sei denn, sie sind Klone eines anderen Objektes
- Snap! macht diese abstrakte Oberklasse “Sprite” jedoch nicht sichtbar; wir sehen nur **die konkreten Objekte** in der **Objektpalette** unten rechts (Ausnahme: die Leinwand!)

Objekte
Objekte werden in der Objektpalette unten rechts in Snap! gelistet. Objekte können eigene **Codeblöcke, Variablen,** und **Skripte** haben, die nur für sie sichtbar sind. Darüber hinaus verfügen Objekte über Kostüme, die sie tragen und Klänge, die sie abspielen können.

Vererbung
Snap! kennt keine Klassen. Das bedeutet, dass eine klassenbasierte Vererbung, wie beispielsweise in Java, in Snap! nicht existiert. Snap! unterstützt jedoch eine **prototypenbasierte** Form der Vererbung, die sog. **Delegation.** Bei diesem Konzept erben Objekte von anderen Objekten anhand sog. **Slots**:

Slot = Sammelbegriff für Attribute und Codeblöcke (Methoden)
Objekte erben von Objekten - **was** sie erben lässt sich dabei dynamisch bestimmen
Beispiel: Von Clyde, dem Elefant-Prototypen wird ein neues Objekt erzeugt, ein Elefant namens Fred. Anders als sein Prototyp Clyde hat Fred nur **drei Beine.**

- Wenn wir **Fred** nach der Anzahl seiner Beine fragen, so prüft er zunächst, ob er diese Frage beantworten kann
- Das tut er, indem er seine **Slots, die die jeweiligen Werte von Clyde überschreiben,** überprüft
- Fred kennt nämlich nicht den Attributwert des Prototypen, weiß aber, welche seiner **Slots** von diesem abweichen
- Überschreibt einer der Slots das Attribut “Beine” von Clyde, so wie in diesem Fall, so gibt **Fred** die Abweichung vom Prototypen als Antwort (in diesem Beispiel **3**)
- Hätte Fred hingegen keinen **Slot**, der die Anzahl der Beine überschreibt, bedeutet das, dass er die Anzahl Beine vom Prototypen ohne Abweichung übernimmt
- Um die Frage dann zu beantworten, fragt Fred dann seinen Prototypen um Rat

Kurz zusammengefasst aus Sicht eines Objekts:
*“Wenn mir **keine andere Information vorliegt**, gehe ich einfach davon aus, dass ich vom Prototypen **nicht abweiche!**”*

Lösungen:

Tipp:
Es empfiehlt sich, den Schülerinnen und Schülern etwa 10-15 Minuten Bearbeitungszeit zu geben.

Nach Ablauf der Bearbeitungszeit für diese Phase sollten die Ergebnisse **gesammelt und besprochen** werden, um sicherzustellen, dass die gesamte Gruppe mit demselben **Grundwissen weiterarbeitet.**

Es fällt auf: Der Prototyp bewegt sich nicht.

Das ist an sich **kein Problem**. Es ist sogar simpler, wenn die Klone sich bewegen, und der Prototyp nur als **“Schablone”**, bzw. **“Bauplan”** dient, um neue Ameisen nach seinem Vorbild zu erzeugen.

	**Tinkering:** Verändere den Code so, dass der Prototyp unsichtbar wird, sobald sein Code erledigt ist.

Ameisen bewegen sich im echten Leben nicht so wie in unserem Beispiel. **Unsere Ameisen laufen schnurgerade!**

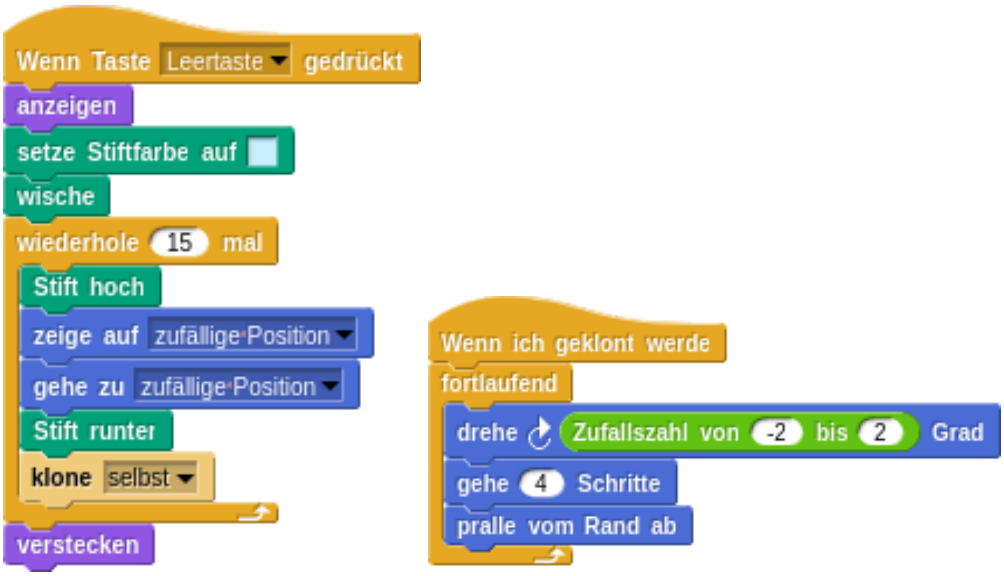
	**Tinkering:** Verändere den Code so, dass Klone sich realistischer bewegen!

 Reflektion

Was haben wir in diesem Abschnitt konzeptionell getan?

Wir haben das Verhalten einer Ameisenkolonie in Snap! simuliert. Dabei haben wir zunächst die Bewegungen einzelner Ameisen implementiert, und dann dafür gesorgt, dass Ameisen eine Spur hinterlassen. Schließlich haben wir uns auf Prototypen und Klone gestützt, um das Klassen-Objekt-Paradigma nachzustellen, denn Snap! kennt keine Klassen. Stattdessen haben wir eine prototypische Ameise beschrieben, und Klone von ihr erzeugt, um neue Ameisen zu erstellen.

Mögliche Lösungen für die Tinkeringphasen:



Ameisen hinterlassen eine Pheromonspur. Dafür sorgt unser Code bereits. Aber eins fehlt:
Der Pheromonspur sollen andere Ameisen nun folgen.

Um die Situation zu veranschaulichen
wechseln wir kurz in ein **neues Snap!-Projekt.**

Aufgabe: Speichere dein bisheriges Ameisen-Projekt!

Aufgabe: Erstelle ein **neues Projekt:**

- **gib** dem **Standardobjekt** den **Namen “Auto”!**
- **zeichne** ein **neues Objekt namens “Bahn”.**
  Das Objekt könnte beispielsweise so aussehen:

(Tipp: Pinselstärke ca. 10!)

- **zeichne** dem **Auto-Objekt** ein neues **Kostüm**, das weniger nach "Standard-Dreiecks-Kostüm" aussieht und mehr nach der **Karosserie eines Autos!**

  (Tipp: Male einen Pfeil auf das Auto, damit du die Blickrichtung leicht erkennen kannst!)

- Setze das **Auto-Objekt** auf der Leinwand auf die **Bahn.**

Unser Ziel ist es, dafür zu sorgen, dass das Auto der Bahn folgt.

Einfache Lösung: Das Auto soll sich drehen, je nachdem welche Pfeiltaste der Benutzer drückt.

Aufgabe: Gib dem Auto Code, der bewirkt, dass der Benutzer es mithilfe der Pfeiltasten drehen und bewegen kann!

Grundsätzlich sind zwei Lösungen für die Bewegung des Autos möglich:

1)

→ Intuitive, elegante Lösung, basiert auf **ereignisbasiertem Paradigma.**

2)

→ Weniger intuitive Lösung, dafür **flüssigere Bewegung.**

So weit, so gut. Wir können wir mithilfe der Pfeiltasten dafür sorgen, dass das Auto auf der Fahrbahn bleibt - können wir auf diese Weise auch dafür sorgen, dass die Ameise der Pheromonspur folgt?

Aufgabe: Können wir unsere Lösung auf das Ameisenprojekt übertragen?

_____

_____

An sich lässt sich die Lösung natürlich problemlos übertragen. Doch wie skalierbar ist sie?

Um eine Ameise kann sich der Benutzer noch kümmern. **Möglicherweise sogar um zwei Ameisen gleichzeitig.**

Doch was, wenn wir nicht nur zwei Ameisen haben, sondern 10? 15? 100?

Offensichtlich ist unsere Lösung noch nicht optimal.

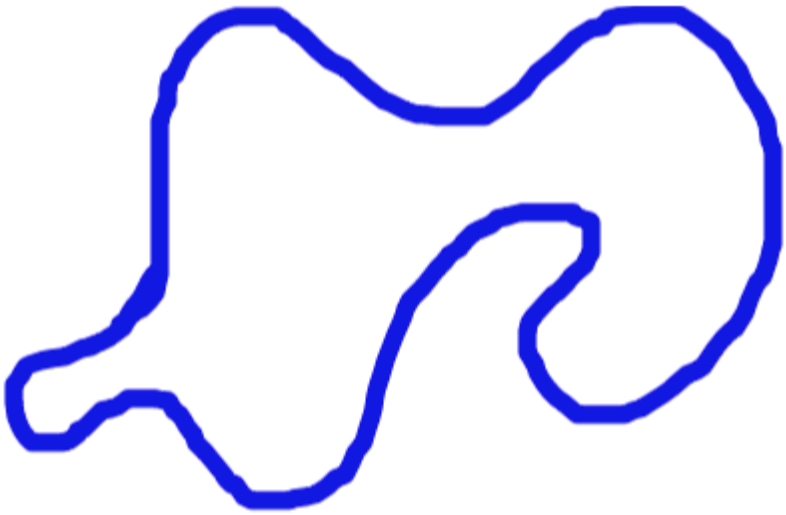
Tinkering: Verändere den Code so, dass das Auto der Bahn folgt - **ohne dass der Benutzer eingreifen muss!**

Möglicherweise basiert deine Lösung darauf, das Auto so fahren zu lassen, dass es immer die Bahn berührt, indem du die Blöcke 

und 

mit entsprechenden Werten verwendest.

Aber funktioniert deine Lösung auch mit anderen Bahnen, wie z. B. dieser?



Falls nicht, ist deine Lösung möglicherweise nicht **allgemein genug!**

Versuche, das Auto **drehen zu lassen, sobald es die Bahn nicht mehr berührt!**

Unsere Lösung muss **allgemein** funktionieren, da wir **nicht vorhersagen können,** wie später im **Ameisenbeispiel** die **Pheromonspur aussehen wird!**

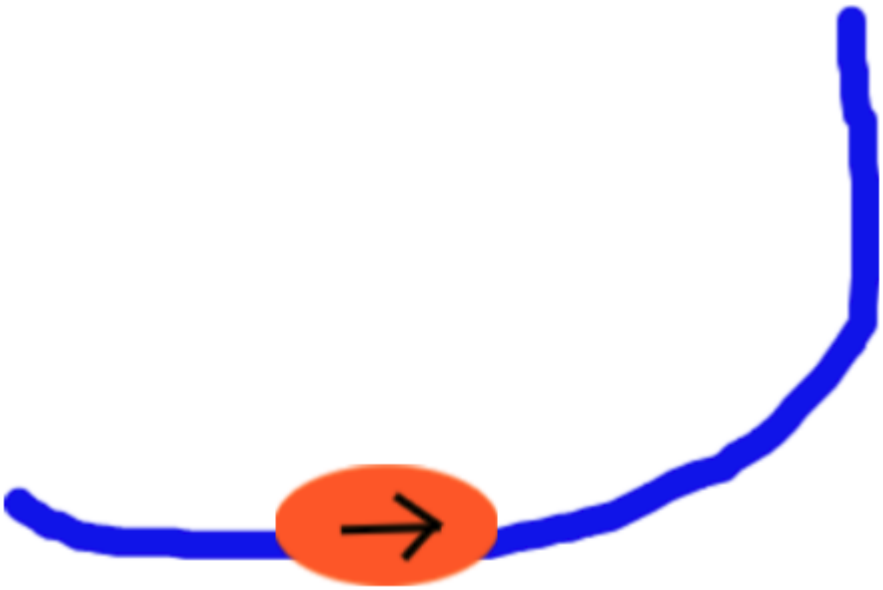
Lösung:

Die Steuerung des Autos lässt sich natürlich ebenso auf die Ameise übertragen, doch ist wenig praktikabel; der Benutzer kann sich zwar um eine Ameise kümmern, doch nicht um eine ganze Kolonie.

Tipp:
Die Schülerinnen und Schüler sollen in der Tinkeringphase selbst ausprobieren, das Problem zu lösen. Sie werden dabei **schrittweise** in der Entwicklung ihrer Lösung **unterstützt** und **geführt**.

… Frustriert?

Sehen wir uns einmal eine Beispielsituation an:



Wo liegt das Problem?

Das Auto kann nicht erkennen, in welche Richtung es sich drehen muss! **In diesem Beispiel** müsste es sich einfach **nach links drehen, sobald es die Rennbahn verlässt. Aber was, wenn die Rennbahn eine Rechtskurve macht?**

Wir müssten also vorhersagen können, wie die Rennbahn verläuft. **Dann wissen wir rechtzeitig, wie das Auto sich drehen muss, um auf der Bahn zu bleiben!**

Hast du eine Idee, wie wir das lösen können?
Wie lösen selbstfahrende Autos im echten Leben dieses Problem?

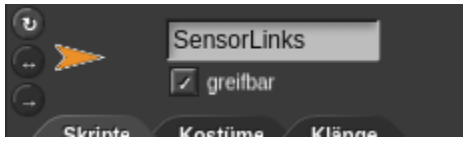
🛠	**Tinkering:** Schaffst du es, das Auto so zu erweitern, dass es ermitteln kann, wie die Rennbahn verläuft?

Die **eleganteste** Lösung sind **Sensoren** am Auto, die den Streckenverlauf erkennen können, und dem Auto weitergeben, in welche Richtung es sich drehen muss.

Solche Sensoren können wir in Snap! ganz einfach selbst bauen.

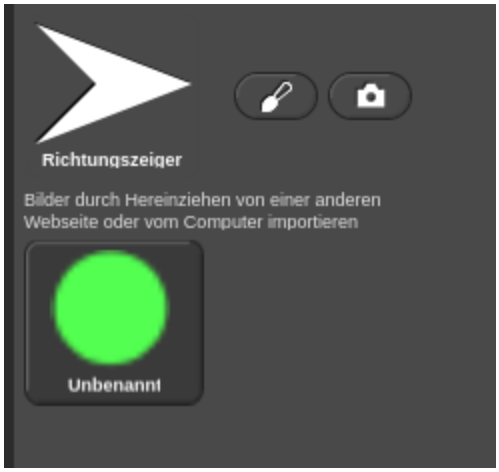
1)

2)



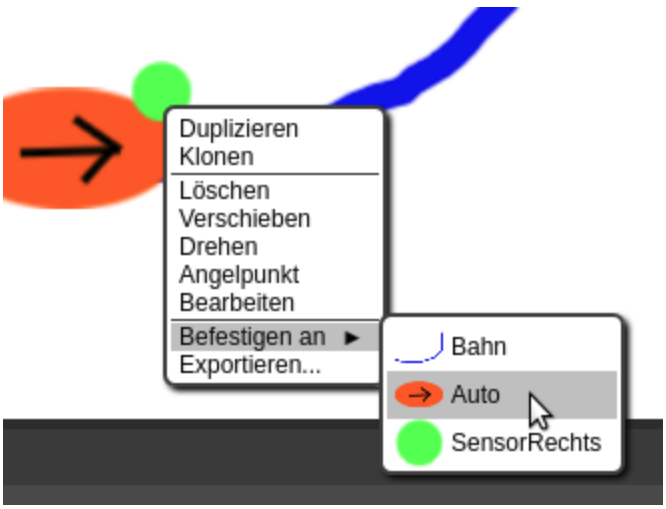
3)

4)



Aufgabe: Oben siehst du, wie das Objekt **SensorLinks** erzeugt wird.
Erzeuge auf die gleiche Art und Weise einen rechten Sensor!

Nun haben wir Sensoren erzeugt, die erkennen sollen, wie die Rennbahn verläuft.
Als nächstes müssen wir sie am Auto befestigen.



Lege die Sensoren auf der Leinwand **auf das Auto**,

Rechtsklick auf die **Sensoren** auf der Leinwand,

Wähle aus diesem Menü **“Befestigen an”** und **“Auto”**

Aufgabe: Befestige auf diese Art und Weise den linken und den rechten Sensor am Auto!
Achte darauf, dass die Sensoren an der jeweils richtigen Seite des Autos sitzen!
Um zu überprüfen, welcher Sensor welcher ist, mach einen **Doppelklick** auf das jeweilige **Sensor-Objekt** und achte auf die **gelbe Hervorhebung!**

Hat alles funktioniert?
Lass das Auto folgendes Skript ausführen, und beobachte, ob sich die Sensoren mitdrehen:

Wenn du alles richtig gemacht hast, sollte sich das Auto gemeinsam mit seinen Sensoren drehen!

Wie benutzen wir nun die Sensoren richtig?

Wenn die Bahn berührt wird, ...

…, dann soll ...

...die Verankerung…
(das Auto!)

… sich um 5 Grad drehen.

↓

Aufgabe: Erstelle ein gleichwertiges Skript für den anderen Sensor!

Beide Sensoren verwenden dabei das Objekt “**Verankerung”**.
Das ist das Objekt, an dem der Sensor angeheftet/verankert ist, d.h. **die Karosserie des Autos.**

Streng genommen sind die Sensoren in diesem Beispiel nicht “nur” Sensoren - denn sie schlagen nicht nur Alarm, wenn die Rennbahn berührt wird, sondern veranlassen auch, dass das **Auto-Objekt** sich dreht.

Somit bleibt nur noch eine Aufgabe für die Karosserie des Autos übrig - das Fahren!

Aufgabe: Gib dem Auto ein Skript, das bewirkt, dass es geradeaus fährt, sobald die grüne Flagge geklickt wurde.

Wir haben das Problem also dadurch gelöst, dass wir das Auto nicht als ein einziges Objekt, sondern als Zusammensetzung (= Kompositum) mehrerer Objekte modelliert und implementiert haben.

Die einzelnen Bestandteile sind dabei individuelle Objekte, die ihren eigenen Code ausführen, sich jedoch als ein großes Objekt auf der Leinwand bewegen.

Aufgabe: Beschreibe, welches Problem **Nesting** in diesem Fall gelöst hat.

_____

_____

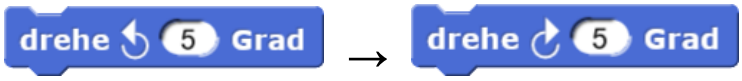
_____

_____

Aufgabe:
Übertrage das **Nesting,** das wir beim Auto angewendet haben, auf das **Ameisenbeispiel!**

Lösungen:

Das Skript ist praktisch identisch, doch der **Drehwinkel muss invertiert werden:**



Teilweise verwenden Schülerinnen und Schüler hier auch 

, was zwar weniger elegant, aber natürlich **nicht falsch** ist.

Das Skript für das Auto ist sehr kurz:

Nesting hat in diesem Fall das Problem gelöst, dass das Auto als individuelles Objekt nicht erkennen kann, in welche Richtung es sich drehen muss, um auf der Spur zu bleiben. **Durch die Zerlegung des Objekts in kleinere Einzelteile (Sensoren) erreichen wir, dass die jeweiligen Einzelteile überprüfen können, ob eine Drehung notwendig ist.**

Reflektion

Was haben wir in diesem Abschnitt konzeptionell getan?
Im letzten Schritt wollten wir erreichen, dass Ameisen den Pheromonspuren anderer Ameisen folgen. Die ideale Lösung für dieses Problem war das sog. **Nesting**, d.h. die Zerlegung der Ameise in Einzelteile. Mithilfe dieses Konzeptes war es möglich, zu überprüfen, in welche Richtung die Ameise sich drehen muss, um der Spur zu folgen.