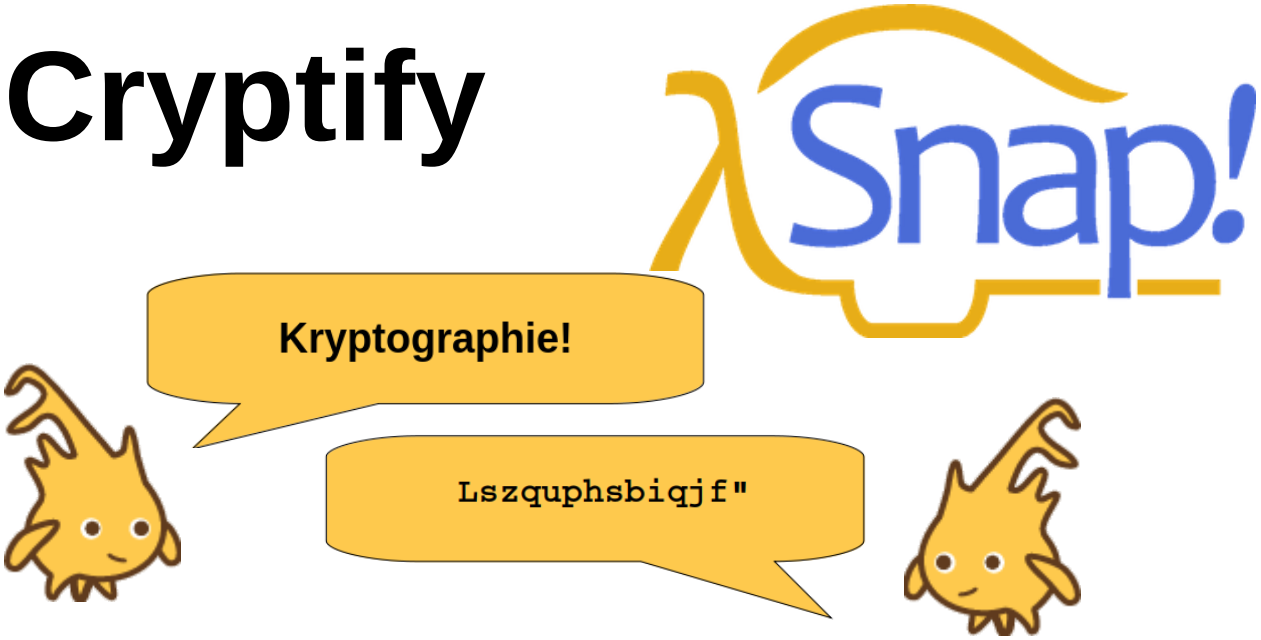
Dieses Beispiel baut auf den Kapiteln
Blockpalette und **First-class Objekte** auf!

Cryptify



In diesem Beispiel soll eine Ur-Form der Verschlüsselung, der sog. **Cäsar-Chiffre** implementiert werden. Es eignet sich besonders gut als Teil einer Einführung in die Kryptographie, beispielsweise in der gymnasialen Oberstufe.
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 Bedienkonzepte, die blockbasierte Sprachen uns ermöglichen - wie etwa **Tinkering** und **Direct Drive**.

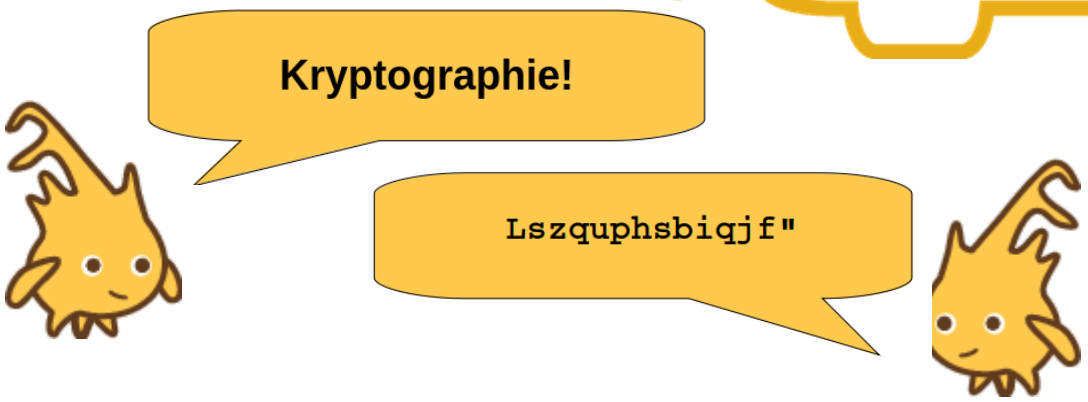
Das Material besteht aus **Schülermaterialien** und **zusätzlichen Anmerkungen für Lehrkräfte** zur Umsetzung im Unterricht sowie **weiterführenden Erklärungen** und **Lösungen.**

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

Cryptify

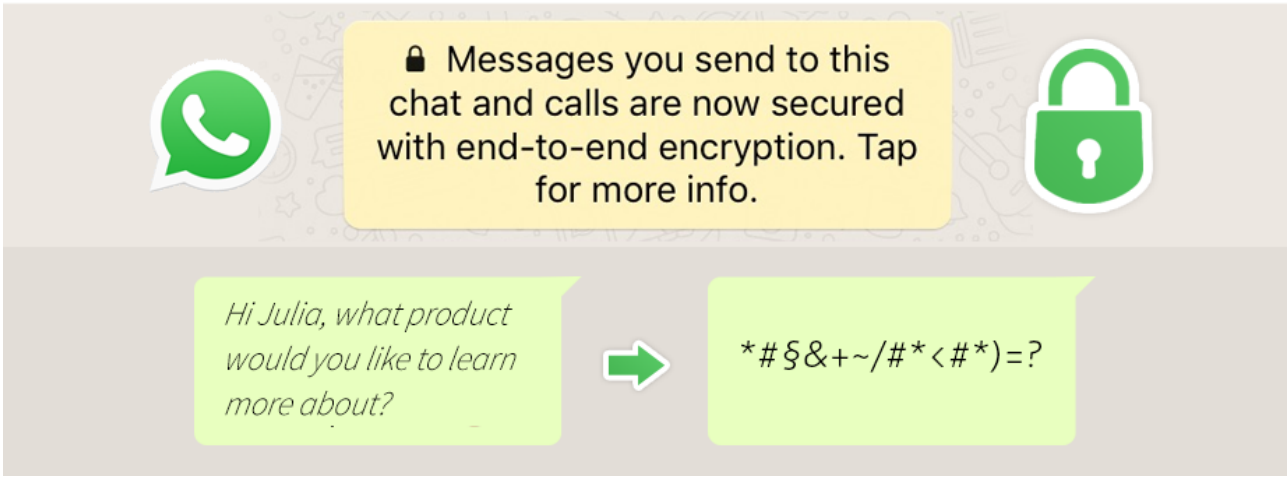


In diesem Modul ver- und entschlüsseln wir geheime Botschaften!

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

Link zum Projekt: bit.ly/snap-crypt

Seit 2016 werden Nachrichten in WhatsApp und Snapchat verschlüsselt:



In Instagram (noch) nicht.

Was bedeutet das? Wie funktioniert diese Art der Verschlüsselung?
Macht das unsere Nachrichten sicherer? Sind unsere Nachrichten damit “abhör-sicher”?

… Warum ist das wichtig?

Um das herauszufinden werden wir in diesem Modul eine Verschlüsselung in Snap! implementieren!

Dazu können wir **Funktionen** verwenden.
Sie haben **Eingabeparameter**, **verarbeiten** diese und liefern dann eine **Ausgabe** zurück.

!

- In Snap! können wir (Funktions-)blöcke einfach ausführen, **indem wir sie anklicken**
- Dieses Bedienkonzept nennen wir **direct drive!**

In der Kategorie 

, bzw. 

finden wir vorwiegend Funktionen und Prädikate.

Nehmen wir zunächst einen der simpelsten Blöcke dieser Kategorie, wie z. B. 

.

Tinkering: Klicke auf den Block, um ihn auszuführen.
Was kannst du dabei beobachten? Experimentiere mit verschiedenen Eingaben!

Aufgabe: Identifiziere die einzelnen Bestandteile von 

!

Eingabe(-n): _____

Verarbeitung: (Operation) _____

Ausgabe: _____

Natürlich handelt es sich hierbei um einen **sehr simplen** Block. Nichtsdestotrotz können wir an ihm wichtige Elemente von Funktionen erkennen.

Außerdem ist der Block für die Verschlüsselung wichtig!

Bonusaufgabe: Durchstöbere die Blockpalette. Welche weiteren Blöcke **könnten wir nutzen**, um eine Verschlüsselung zu implementieren?

Hast du eine Idee? Vielleicht sogar mehr als eine?
Dann blättere um auf die nächste Seite.

Tipps:
Snap! kennt drei Arten von Blöcken:

1. 

Befehlen, bzw. *commands*: Diese Blöcke führen typischerweise Aktionen aus, die eine **sichtbare** Auswirkung auf das Projekt haben. Oftmals sehen wir diese Auswirkungen auf der **Leinwand. Befehle haben keine Rückgabe!**

2. 

Funktionen, bzw. *reporters:* Diese Blöcke führen typischerweise Berechnungen oder Abfragen aus. Sie haben nur selten eine **sichtbare** Auswirkung auf das Projekt. Dafür liefern diese Blöcke **Rückgabewerte!**

3. 

Prädikaten, bzw. *predicates:* Wie Funktionen haben auch Prädikate Rückgabewerte. Obwohl der Rückgabewert eines Prädikats prinzipiell ein beliebiger Datentyp, z. B. ein Integer oder ein String sein kann, ist ihr Sinn ein anderer: sie liefern typischerweise boolesche Werte, d.h. **true** oder **false** zurück, und werden oftmals zur Steuerung des Programmablaufs benutzt.

Tipp: Hier könnte es im Unterricht sinnvoll sein, einen Vergleich zum Konzept der **Methoden** oder **Funktionen** aus bereits bekannten Programmiersprachen zu ziehen. Dadurch wird deutlich, dass Blockformen konzeptionell nichts anderes sind, als was Schülerinnen und Schüler bereits aus anderen Sprachen kennen.

Lösung:
Eingabe(-n): 5 und **3**
Verarbeitung: Addition
Ausgabe: 8

Tipp: An dieser Stelle bietet es sich im Unterricht an, das in der Informatik grundlegende Prinzip von Eingabe-Verarbeitung-Ausgabe (EVA) zu thematisieren, bzw. aufzugreifen. Es könnte außerdem gewinnbringend sein, ein Äquivalent des obigen Blocks in einer bereits bekannten Programmiersprache zu zeigen.

Ein heißer Tipp ist der Block 

.

Tinkering: Klicke auf den Block, um ihn auszuführen.
Was kannst du dabei beobachten? Experimentiere mit verschiedenen Eingaben!

Das Gegenstück zu diesem Block bildet der Block 

.

Tinkering: Klicke auf den Block, um ihn auszuführen.
Was kannst du dabei beobachten? Experimentiere mit verschiedenen Eingaben!

Aufgabe: Kombiniere die drei Blöcke

, 

, und 

,

sodass das Skript am Ende **“b” als Rückgabe liefert!**

Hierbei handelt es sich bereits um eine **Verschlüsselung!** Einzelne Buchstaben werden dabei linear nach rechts **verschoben**, d.h. aus “a” wird “b”, aus “b” wird “c”, usw. Ein Beispiel für Verschlüsselung durch Verschiebung ist die **Cäsar-Chiffre**.

In diesem Beispiel betragen die Eingabewerte “a” und 1.
Den zweiten Eingabewerte bezeichnen wir im Folgenden als **Verschiebung.**

Aufgabe: Verändere die Eingabewerte, sodass:

- das Skript die Eingabe “a” nimmt und als Rückgabe “f” liefert

  Notiere die Verschiebung! _____

- das Skript die Eingabe “z” nimmt und als Rückgabe “Z” liefert

  Notiere die Verschiebung! _____

- das Skript die Eingabe “z” nimmt und als Rückgabe “A” liefert

  Notiere die Verschiebung! _____

Lösungen:

, Verschiebung = 5

, Verschiebung = -32

, Verschiebung = -57

Mithilfe dieses Skripts können wir nun einzelne Buchstaben verschlüsseln.

Leider sieht der Block nicht sehr ansprechend aus und ist etwas unhandlich.

Um diese Probleme zu lösen werden wir nun einen **eigenen Block** für unsere Verschlüsselung erstellen.

Aufgabe: Erzeuge einen **neuen Block**. Der Block soll:

- ein Funktionsblock sein
- einen passenden (sprechenden) Namen tragen
- zwei Eingabeparameter haben: ein Wort, und eine Verschiebung
- einer passenden Kategorie angehören (beispielsweise Operatoren)

In diesen neuen Block können wir nun unser bisheriges Konstrukt hineinziehen. Dabei müssen wir darauf achten, die Eingabewerte, die wir bisher manuell eingegeben haben, durch die **Eingabeparameter des eigenen Blocks zu ersetzen.**

Aufgabe:

- Ziehe das 

-Konstrukt in den eigenen Block
- Ersetze die Zahlen und Buchstaben durch die Eingabeparameter des Blocks

Klicke auf **OK**.

In der entsprechenden Blockkategorie findest du nun deinen Block, beispielsweise 

Tinkering: Teste deinen Block mit verschiedenen Eingaben!
Was passiert, wenn du Wörter wie “Hallo” verschlüsseln möchtest?

Lösung:


+ shift + letter + by + number +
berichte Unicode Unicode Wert von letter + number als Buchstabe

Momentan funktioniert unsere Verschlüsselung so, dass einzelne Buchstaben ganz einfach **um einen bestimmten Wert verschoben werden.**

Mit Wörtern funktioniert das leider nicht, denn sie bestehen aus mehreren Buchstaben.

Unsere Lösung:
Wir wenden unsere **Verschiebungs-Verschlüsselung**
auf jeden Buchstaben eines Wortes an.

In Snap! bedeutet das:

1) **Wir machen aus dem übergebenen Wort eine Liste**

2) **Wir wenden auf jedes Element dieser Liste einen bestimmten Block an …**

3) **… und zwar unseren Verschiebungs-Block**

Aufgabe: Kombiniere die drei Blöcke, und verschachtle sie so, dass das Skript das Wort “aaa” zu “ddd” verschlüsselt!

(Tipp: lass den ersten Parameter des Verschiebungsblocks frei. Dadurch nimmt der Block als Parameter jedes einzelne Element der übergebenen Liste)

Der Rückgabetyp ist momentan noch eine Liste der einzelnen Buchstaben. **Mit einer einzigen letzten Anpassung können wir aus dieser Liste wieder ein Wort machen:**

Als letzten Schritt wollen wir das alles nun in einem einzigen, praktischen Block verpacken!

Tipps:

Lösung:

Aufgabe: Erzeuge einen **neuen Block.** Der Block soll:

- ein Funktionsblock sein
- einen passenden (sprechenden) Namen tragen
- zwei Eingabeparameter haben: ein Wort, und eine Verschiebung
- einer passenden Kategorie angehören (beispielsweise Operatoren)

Ziehe das 

-Konstrukt in den eigenen Block und ersetze die Zahlen und Buchstaben durch die Eingabeparameter des Blocks.

Dieser Block **verschlüsselt** nun ein ***Eingabewort***
mithilfe eines übergebenen ***Schlüssels*** **(das ist unsere Verschiebung!).**

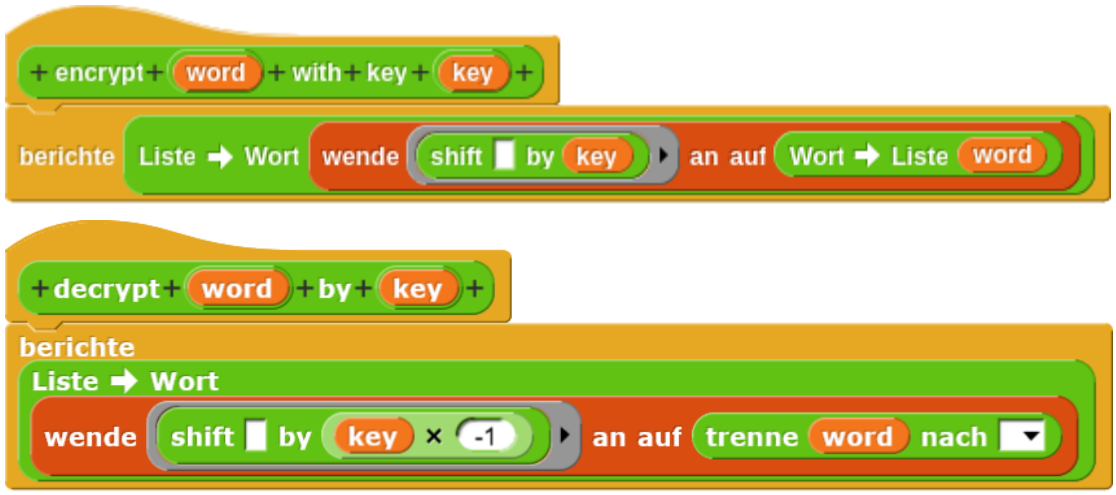
Können wir die Verschlüsselung auch wieder rückgängig machen, sodass wir geheime Botschaften entschlüsseln können, wenn wir den Schlüssel kennen?

Aufgabe: Erzeuge einen **neuen Block.** Der Block soll:

- ein Funktionsblock sein
- einen passenden (sprechenden) Namen tragen
- zwei Eingabeparameter haben: ein Wort, und eine Verschiebung
- einer passenden Kategorie angehören (beispielsweise Operatoren)

Passe den Block so an, dass er ein Wort mithilfe eines übergebenen Schlüssels **entschlüsselt** und das Wort in Klartext zurückgibt!

Lösung:


+ encrypt + word + with + key + key +
berichte Liste ➜ Wort wende shift by key an auf Wort ➜ Liste word
+ decrypt + word + by + key +
berichte
Liste ➜ Wort
wende shift by key × -1 an auf trenne word nach

Reflektion

Was haben wir in diesem Abschnitt konzeptionell getan?
Wir haben eine der wichtigsten Arten von Verschlüsselung, die sog. **Cäsar-Chiffre,** in Snap! nachgestellt. Dabei haben wir uns auf Funktionsblöcke und direct drive (die Möglichkeit, Code anstelle einer main-Methode einfach durch Mausklick auszuführen) gestützt. Nach jedem Zwischenschritt haben wir unsere Blöcke ausprobiert, und dann in weiteren Blöcken verschachtelt, um neue Funktionalitäten hinzuzufügen.

Schließlich haben wir so eine einfache Verschlüsselung erzeugt, die durch Benutzung desselben Schlüssels auch wieder rückgängig gemacht werden kann. Durch die Möglichkeit von Snap!, Variablen zu exportieren, kann ein “Schlüssel” auch physikalisch abgespeichert werden, beispielsweise auf einem USB-Stick.