Mobile CSP | Student Lesson

Unit 6 | *Caesar Cipher Tutorial*

Course Listing: http://course.mobilecsp.org

This lesson introduces the Caesar Cipher, one of the earliest and simplest examples of a ***substitution cipher*** in the field of cryptography. The Caesar Cipher app will encrypt and decrypt secret messages by shifting the letters in the alphabet by a certain shift number..

Objectives:

In this lesson you will :

- learn basic concepts about cryptography and the Caesar cipher,
- build an app that implements Caesar cipher encryption and decryption,
- learn how to use local variables (as opposed to global variables) in an app,
- learn how to use a function (a procedure that produces a value) in an app.

Short Handout

Overview

In this lesson we will build an app that performs Caesar-cipher encryption and decryption of messages. Before jumping into the app itself, we begin with a brief introduction to *cryptography* and to the Caesar cipher algorithm.

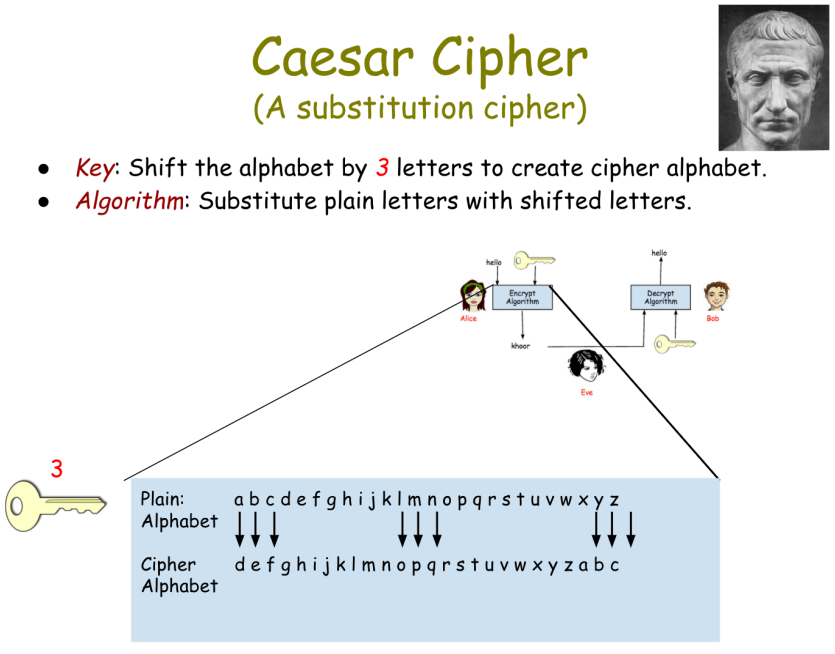
Cryptography

Cryptography means secret writing. It is the art and science of sending secret messages and it has been used by generals and governments and everyday people practically since the invention of written language. Modern cryptographic techniques are essential to guaranteeing the security of our transactions on the Internet.

Cryptography plays a role whenever you make an online purchase at Amazon or provide your password to Google. Whenever you see the https protocol in your browser, you can rest assured that your communications are secure because they are being encrypted with strong, unbreakable encryption. If we couldn't rely on those transactions being secure we really wouldn't have the Internet as we know it today.

Video: The Caesar Cipher

In the next lessons, we will look at several different versions of cryptography, including the strong encryption that protects our Internet transactions. But let's begin here with a simple cipher, the Caesar cipher, so named because it was used by Julius Caesar in the 1st century B.C. The following video will explain the basics of the Caesar cipher. Click on the picture or link to watch this presentation on Caesar Cipher.



(Click the image to start the video.)

Some Exercises

Before proceeding, try the following set of exercises to make sure you understand how Caesar cipher works. For the first two exercises, use the following Caesar alphabet, which has a key (a shift) of 3. Try to do these exercises by hand.

PLAIN_ALPHABET: **`abcdefghijklmnopqrstuvwxyz`**
CIPHER_ALPHABET: **`DEFGHIJKLMNOPQRSTUVWXYZABC`**

1. By Hand: Use the Caesar cipher shift of 3 to encrypt your name. Then use the Caesar Cipher Widget to check your answer.
2. Encrypt a short message for your partner using the cipher_alphabet with shift 3 above. Trade the encrypted messages and decrypt them.
3. Create the CIPHER_ALPHABET that would result from a Caesar shift of 5.
4. Do the self-check exercises after this section online.

Functions and Local Variables

This lesson introduces two new programming concepts, local variables and functions. A ***local variable*** (in contrast to a global variable) is one that has a ***limited scope***, which means that it only exists and can only be used within a block of code, for example in a procedure or a function. A ***global variable*** can be used anywhere within the program -- or, at least in App Inventor, anywhere within the blocks workspace associated with Screen1. A good place to use local variables is in procedure and function definitions.

A ***function*** is a procedure that returns a value. A simple example would be the *sqrt()* function -- when you call *sqrt(25)* it will return the value 5, which can then be used in other expressions. For example, 5 + sqrt(36) is 11. You can't do this with procedures that don't return a value.

The Caesar Cipher App: Getting Started

To get started click on this link to open App Inventor and import the CaesarCipherTemplate. Use the *Save As* button to rename your project "CaesarCipher".

The User Interface

The app's User Interface is already designed for you. This walkthrough will go over the app's

main elements.

The UI for our app consists of the following components: two ***TextBoxes*** are used for inputting the plaintext message ('Plaintext') or the ciphertext message (QMBJOUFYU) respectively. A third *TextBox* is used to input the shift (2).

Two ***Buttons*** are used for encrypting and decrypting. When the 'Encrypt' button is clicked, the message in the plaintext TextBox ('Plaintext') is encrypted using the shift (2 in this case) and the result is displayed in the the ciphertext TextBox ('QMBJOUFYU'). When the 'Decrypt' button is clicked, the message in the ciphertext TextBox should be *decrypted* using the shift (2) and the result displayed in the plaintext Textbox.

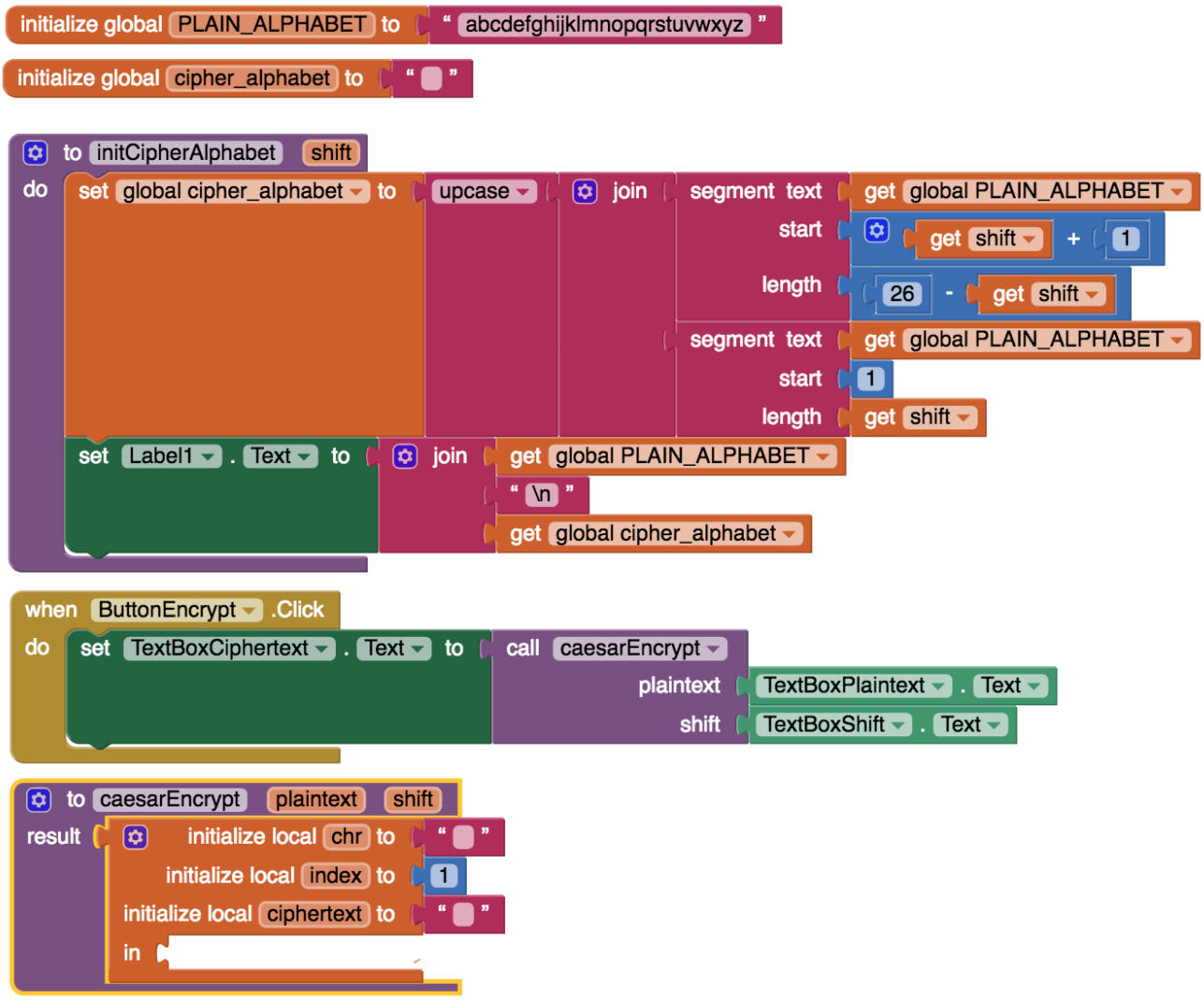
Finally, there are two ***Labels***. One ('Shift:') is used as a prompt to show the user what to type into the shift Textbox. The other ('Text for Label') is used to help us debug the app as we are developing it. It will be hidden from the user once we have finished the app.

In addition,

- The Screen's orientation property has been set to 'Portrait'.
- The *TextBoxShift.NumbersOnly* property has been set to true, which means that only numbers can be input into that element.

Coding the App

The template for this app already has a lot of the code built for you. The following screenshot shows the entire template workspace.

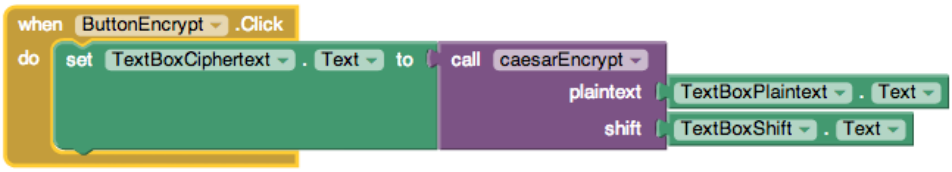


Variables

There are just two ***global variables*** necessary for this app, the *PLAIN_ALPHABET* and the *cipher_alphabet.* Following our coding convention, the *PLAIN_ALPHABET's* is all UPPERCASE because it is a constant -- its value will never change. The *cipher-alphabet* will change based on the shift, so its name is in lowercase.

Events and Event Handlers

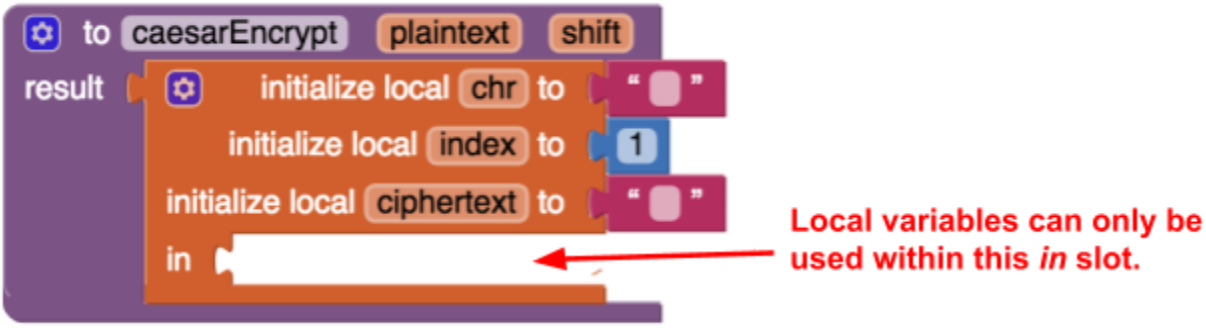
There are just two events that must be handled by this app: the 'Encrypt' and 'Decrypt' Click events. Let's look at the code for the *ButtonEncrypt.Click* handler:



When *ButtonEncrypt* is clicked, it calls the ***caesarEncrypt*** function, which encrypts the string that the user input into the *TextBoxPlaintext* component and displays the result in the *TextBoxCiphertext* component. This will cause the function's result, or value, to be displayed in the user interface. Notice how the plaintext and the shift are taken directly from their respective *TextBoxes*, By using a ***function*** (rather than a procedure) this operation can all be done with one statement.

The *caesarEncrypt* Function

Rather than coding all of the encryption details in the event handler, we are using the ***caesarEncrypt*** function to perform the encryption for us. Here is a ***stub*** version of the function definition. You will code the implementation.



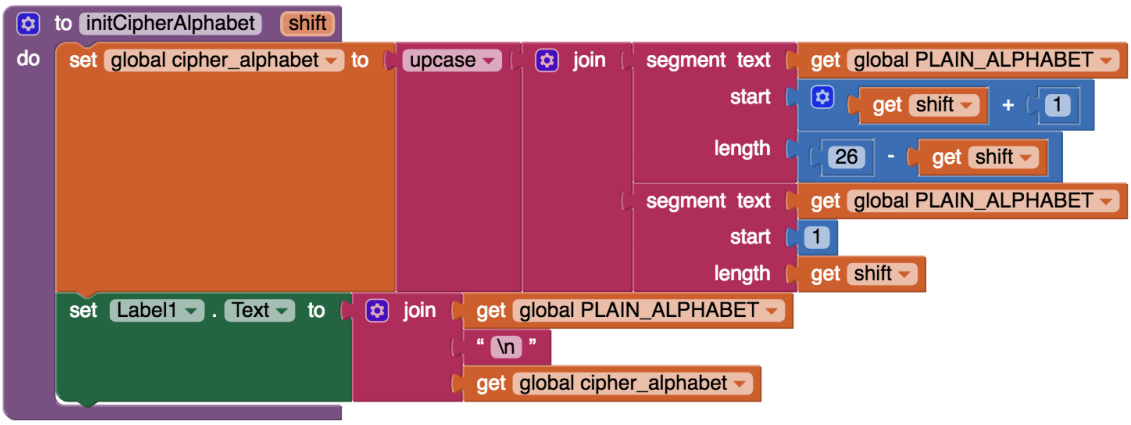
The function takes two ***parameters,*** the ***plaintext*** message that we want to encrypt and the ***shift*** to use in setting up the cipher alphabet. As a ***result*** the function will return the encrypted message. Note the use of the three ***local variables:*** The ***chr*** variable will store individual characters that are pulled from the *plaintext* message. The ***index*** variable will store the *chr's* index in the *PLAIN_ALPHABET* and the ***ciphertext*** variable will store the encrypted message that will be returned as a result.

> **Why Local Variables.** As local variables, these three variables have ***limited scope,*** which means they can only be used within the ***in*** slot -- they cannot be used elsewhere in the workspace. One advantage of local variables is that they reduce the number of global variables that you have to manage in a program. Another advantage is that they make it easier to understand the variable's role because it is limited to just that local

scope. Local variables often make debugging easier too because if there is a problem with their values, you only need to check the block of code where they are set up, unlike global variables that can be changed anywhere in the program.

Code Walkthrough: The initCipherAlphabet Procedure

As noted above, as a substitution cipher, the Caesar cipher uses a ***cipher alphabet*** to perform the encryption. We will use the ***initCipherAlphabet*** procedure to construct the cipher alphabet:



The "\n" is a special character to put in a newline so that each alphabet is on separate lines. Recall that the *PLAIN_ALPHABET* stores the 26 letters 'a' through 'z' and to construct the *cipher_alphabet* we want to shift those letters by the *shift* amount, wrapping some letters around to the right-hand-side of the alphabet. Consider the plaintext alphabet written out with index numbers above it:

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 (indexes)
a b c d e f g h i j k l m n o p q r s t u v w x y z (plaintext)

If our shift is 3, then we would shift every letter three spaces to the left and let the first three letters wrap around to the right-hand-side:

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 (indexes)
d e f g h i j k l m n o p q r s t u v w x y z **a b c** (plaintext)

We could use a loop to perform this shift operation. But there is an easier way to do this. Notice that we get the same result if we concatenate the last 23 letters of the alphabet and the

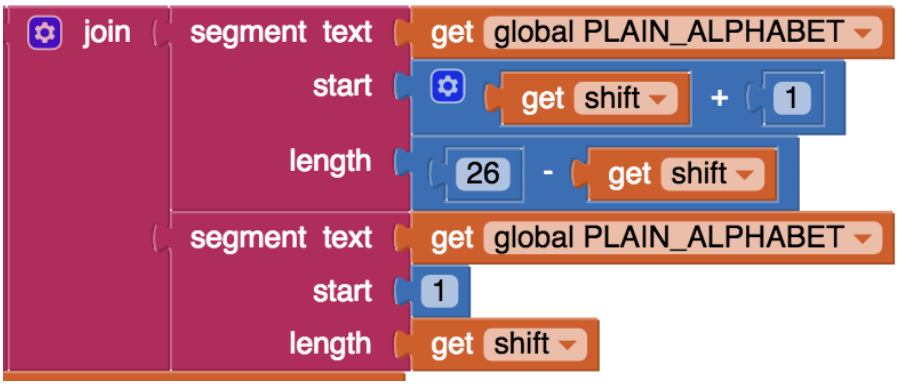
first 3 letters:

d e f g h i j k l m n o p q r s t u v w x y z + **a b c**

So, to construct the cipher_alphabet we want to grab 23 (26 - shift) letters from the PLAIN_ALPHBET, starting at index 4 (shift + 1), and join them to all the letters starting at 1 and going up to index 3. That is, we join DEFGHIJKLMNOPQRSTUVWXYZ and **ABC.** This will give us this pair of alphabets:

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 (indexes)
a b c d e f g h i j k l m n o p q r s t u v w x y z (plaintext)
D E F G H I J K L M N O P Q R S T U V W X Y Z A B C (ciphertext)

To perform this operation in App Inventor we will use the built-in ***segment*** function from the *Text* drawer. As its name suggests, the ***segment*** function grabs a certain segment (or ***substring***) from a piece of text.



The first segment block grabs last 23 letters (26 - shift) from the PLAIN_ALPHABET. Those are the letters defghijklmnopqrstuvwxyz. The second segment block grabs the first 3 (shift) letters from the PLAIN_ALPHABET -- i.e., the letters **abc**. When you join them together you get the cipher alphabet shown above:

defghijklmnopqrstuvwxyz**abc**

Code Walkthrough: The Encryption Algorithm

The encryption algorithm needs to loop through the letters of the plaintext message replacing each plaintext letter with the corresponding ciphertext letter. However, if we did that, we would

be changing the plaintext message itself. So, instead, we will loop through the plaintext message and construct a new message, the ciphertext message, by replacing each plaintext letter with the corresponding ciphertext letter. In ***pseudocode*** we get the following algorithm:

```
initCipherAlphabet(shift)
ciphertext ← ""
FOR EACH letter IN the plaintext
    Find corresponding letter in ciphertext alphabet
    Concatenate corresponding letter to ciphertext
RETURN ciphertext
```

Let's trace through this algorithm with a simple example. Suppose our plaintext message is ***hello*** and our shift is *3.* When we call *initCipherAlphabet* we get the following pair of alphabets:

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 (indexes)
a b c d e f g h i j k l m n o p q r s t u v w x y z (PLAIN_ALPHABET)
D E F G H I J K L M N O P Q R S T U V W X Y Z A B C (cipher_alphabet)

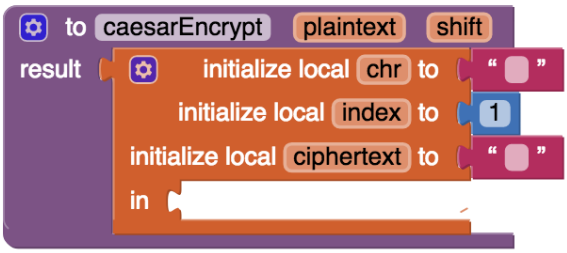
Our ciphertext starts out as "". Then we loop through the letters in the word ***hello.*** For the first letter, 'h', we concatenate its corresponding cipher_alphabet letter, *'K',* to the ciphertext. This gives "K". For the second plaintext letter, 'e', we would concatenate *'H'* from the cipher_alphabet and so on. The following table shows how the ciphertext message will grow from "" (the empty string) to "KHOOR" as the loop progresses:

Iteration	Plaintext letter (chr)	Cipher_alphabet letter	Ciphertext result
0			
1	h	K	K
2	e	H	KH
3	l	O	KHO
4	l	O	KHOO
5	o	R	KHOOR

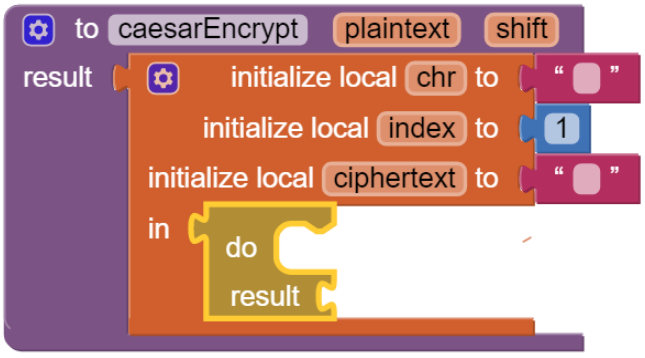
After 5 iterations of the loop the loop will stop and the algorithm will return '*KHOOR'* as the encryption of *'hello'.*

Coding CaesarEncrypt

Your task is to implement this algorithm in App Inventor as the *caesarEncrypt* function. You are given the following stub version of *caesarEncrypt* in the template. Notice it has 3 ***local variables.*** The ***ciphertext*** variable will store the encrypted message, which will be built letter by letter. It is initialized to the empty string. The ***chr*** variable will store the plaintext letter that we need to replace with its cipher_alphabet substitute. And the ***index*** variable will store the index of *chr* in the PLAIN_ALPHABET.

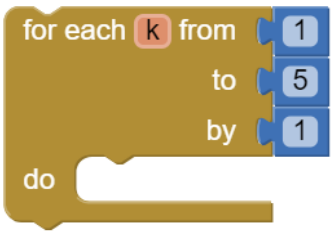


Notice that the "*initialize local in*" block has is looking for a block with a stub. First, put in a *do* block from the Controls drawer so that we can return a result for the function and have a gap where we can put in other blocks.



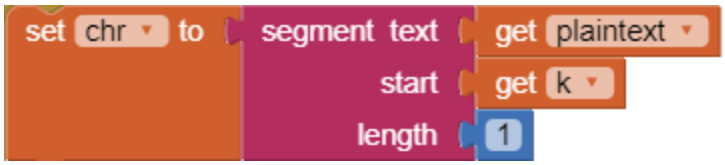
Inside the *do* block, you will first need to call the procedure:

initCipherAlphabet with the **shift** parameter given to the caesarEncrypt function.



Then, you will need a loop to encrypt the plaintext. One problem is that App Inventor does not have a block that is equivalent to ***for-each-letter-in*** the plaintext message. So how will you code that? One way is to use the counting loop ***for-each-number*** block. (In this case I've replaced ***number*** with ***k.*** The ***k*** in this case will be the index of the letter in the plaintext message. So for our example ***hello*** message, the letter at index 1(k=1) is *'h'.* The letter at index 2 (k=2) is *'e'* and so on.

Using this block you would loop from 1 to the ***length*** of the plaintext message by 1. Then inside the loop, you would use the ***segment*** block to get the ***kth*** letter of the plaintext message and assign it to the local ***chr*** variable:



Once you have the ith letter, you need to find where it is -- i.e., its index -- in the PLAIN_ALPHABET. For this task we can use the ***starts-at*** block from the Text drawer. This block gives us the index at which a certain piece of text (a **substring)** starts at in another piece of text. If the substring is not in the larger string, then it will return 0. We can use the block to set the local ***index*** variable:

Now that we know the index of the plaintext letter in the PLAIN_ALPHABET, we can use that index to get the corresponding letter from the *cipher_alphabet* using the ***segment*** block again:

That's the letter that we'll want to concatenate to the ***ciphertext*** variable using a ***join block.***

However, what would happen if the plaintext letter does not occur in the PLAIN_ALPHABET? That would be the case for spaces, numbers, uppercase letters, punctuation and so on. The ***starts-at*** block will return 0 if the plaintext letter is not found in the PLAIN_ALPHABET. In that case, we should just concatenate the plaintext letter itself to the ciphertext message. In other words:

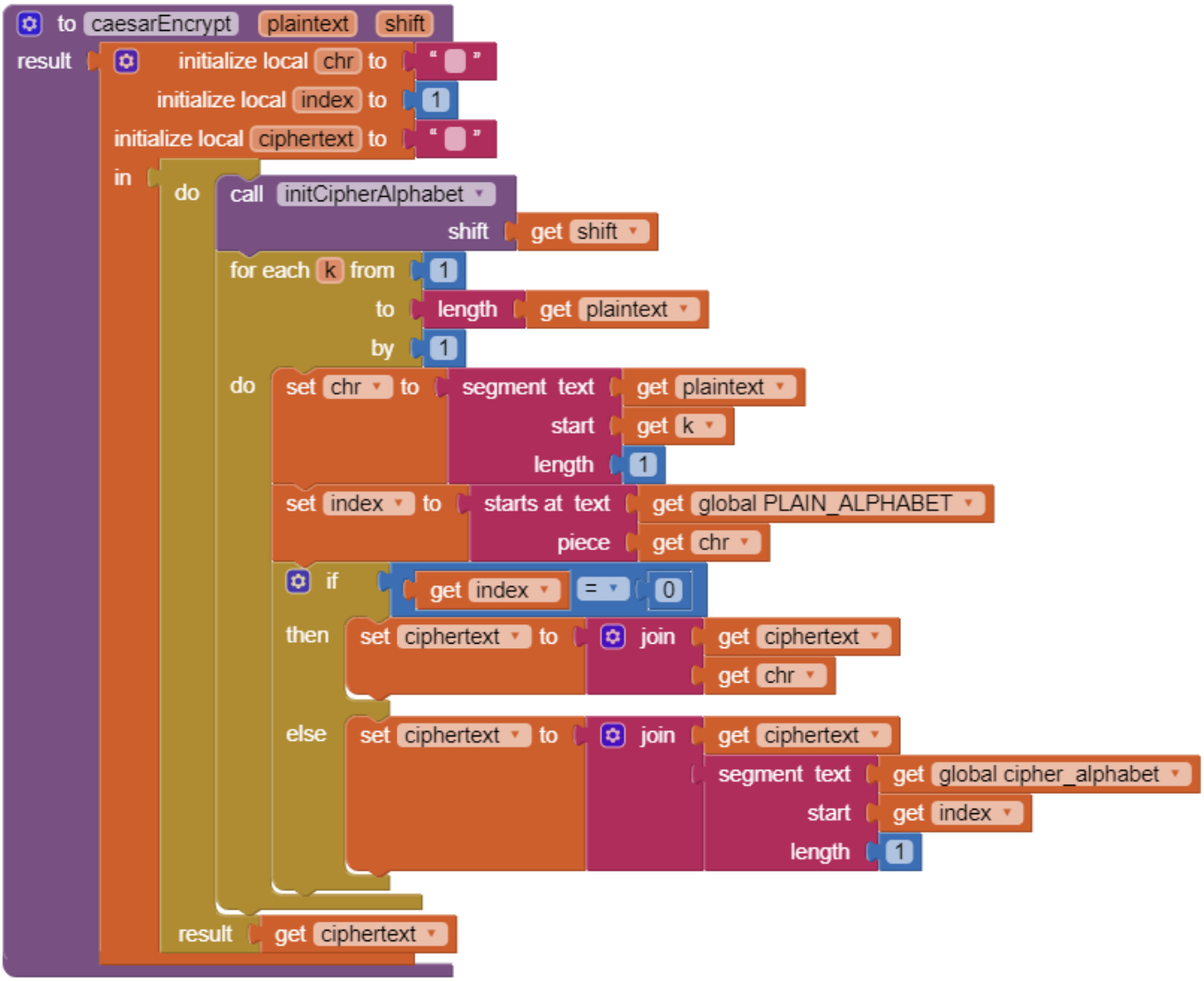
if the ***index*** is 0, we join the plaintext letter to ciphertext, but otherwise.we join the corresponding letter from the *cipher_alphabet.*

Taken together, this leads to the following more detailed pseudocode description of the *caesarEncrypt* function:

```
PROCEDURE caesarEncrypt(plaintext, shift)
{
  chr ← ""
```

```
  index ← 1
  ciphertext ← ""
  initCipherAlphabet(shift)
  FOR EACH k FROM 1 TO length(plaintext)
  {
      chr ← segment(plaintext, k, 1)
      index ← starts-at(PLAIN_ALPHABET, chr)
      IF  (index = 0)
        ciphertext ← join (ciphertext, chr)
      ELSE
        ciphertext ← join(ciphertext,segment(cipher_alphabet,index,1))
   }
   RETURN ciphertext
}
```

Here is the whole code for encrypt:



TO DO

Your task is to code the ***caesarEncrypt*** function and get the app working using Caesar cipher. Here's a summary of what you need to do:

Abstraction: Function	Algorithms
caesarEncrypt	Follow this pseudocode inside the local variables and a do-result block to go through each letter in the plaintext and find the shifted letter in the cipher_alphabet and join it on to the ciphertext which it returns.

```
FOR EACH k FROM 1 TO length of plaintext
{
  chr ← segment plaintext starting at k for length 1
  index ← starts-at(plaintext, chr)
  IF  (index = 0)
      ciphertext ← join (ciphertext, chr)
  ELSE
      ciphertext ← join(ciphertext,
                        segment(cipher_alphabet,index,1))
}
RETURN ciphertext
```

Testing the App

Make sure you test the app thoroughly to make sure the encrypt function is working correctly for various values of the shift. You'll need to figure out the correct encryption by hand or using the widget in order to tell if your implementation is correct.

Inputs	Expected Outputs	Actual Outputs
Type "hello world" into the plaintext box. Set the shift to 3 and click the "Encrypt" button.	The resulting output in the ciphertext box should be "KHOOR ZRUOG".	?

Enhancements and Extensions

1. **Decryption**. Implement the *caesarDecrypt* function and the handler for the Decrypt button to enable the app to perform decryption. Decryption is the mirror image of encryption. Whereas for *encryption*, you replace every character in the *plaintext* with the corresponding letter from the CIPHER_ALPHABET, for *decryption* you go through the *ciphertext* and replace every character with the corresponding letter from the PLAIN_ALPHABET. When you are testing this app, only type in lowercase letters in the plaintext textbox to encrypt, and only type in uppercase letters in the Ciphertext textbox to decrypt.

2. **Extend the Alphabet.** As it is currently implemented, the plaintext alphabet consists only of lowercase letters 'a' through 'z'. This means that digits (0 through 9) and uppercase letters ('A' through 'Z') are not encrypted. That's a security flaw that makes it easier for Eve, the eavesdropper, to break the cipher and discover the secret message.

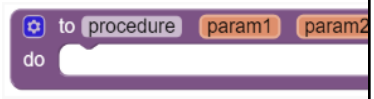
To fix this, extend the plaintext alphabet to include digits and UPPERCASE letters. If you use the appropriate amount of abstraction, this should be a simple change to implement!

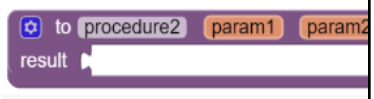
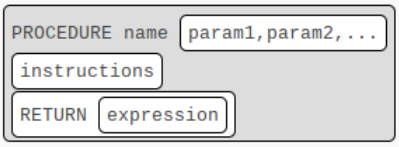
3. **Challenging (Optional).** Preserving the blank spaces between words makes it easier for Eve the eavesdropper to crack the encrypted message. To make this more difficult, write a function that will take a sentence and output the letters in blocks of length 4 with all punctuation (i.e., all characters not in the PLAINTEXT alphabet) removed. For example, the function would take 'this, is a test message!!' return 'this isat estm essa ge'.

Summary

The following compares App Inventor procedures and functions (procedures with return values) to AP pseudocode and blocks.

AI Blocks	AP Pseudocode	AP Pseudoblocks

	PROCEDURE name(param1,param2,...)
	{
	instructions
	}
		

	PROCEDURE name(param1,param2,...)
	{
	instructions
	RETURN (expression)
	}
		

Vocabulary Review

- Function
- Return Value
- Local Variables
- Encryption
- Decryption
- Cryptography
- Cipher

Nice work! Complete the Self-Check Exercises and Portfolio Reflection Questions as directed by your instructor.