Fortune

> `fortune` is a program that displays a pseudorandom message from a database of quotations that first appeared in Version 7 Unix.The most common version on modern systems is the BSD fortune, originally written by Ken Arnold. Distributions of fortune are usually bundled with a collection of themed files, containing sayings like those found on fortune cookies (hence the name), quotations from famous people, jokes, or poetry.
>
> https://en.wikipedia.org/wiki/Fortune_(Unix)

To show some techniques for downloading data from the Internet, transforming it and displaying it, I have chosen to create a version of the ancient unix command line program, fortune.

There are several online databases containing the quotes, so our program will download a random quote and display it.

It used to be common for systems administrators to configure "fortune" to run every time a new terminal session starts up - in fact, I have my Mac configured to do this.

The fortunes have mostly stayed fairly static for the last 30 years or so, so they reflect geeky humour and frequently quote comedians which were popular back then. This is all part of its charm!

Getting data from an API

An API (Application Programming Interface) is like a web page which is designed to be processed by a computer, rather than being displayed in a web browser.

When we get the data, we will actually be doing exactly the same thing as your web browser does, but the data we fetch will be available for processing in our program.

We want to get some data from the remote server, and appropriately enough we will need to use an HTTP[1] GET request to do this.

The API we are going to be using is available at https://www.muppetlabs.de, and the front page of that site shows a random fortune in a web page. At the top right of the page there is an API INFO link which goes to https://www.muppetlabs.de/info, where you can read the technical details.

If you keep clicking the "MORE COOKIE!!" button, you will probably quickly find that the site delivers fortunes in both German and English.

The important part of the page starts where it defines the request as

```
GET /api/?lang=LANG&categories=CATEGORIES
```

It then goes on to tell us that `LANG` can be either `de` (for German) or `en` (for English), and `CATEGORIES` should be a comma separated list of categories chosen from the list towards the end of the page.

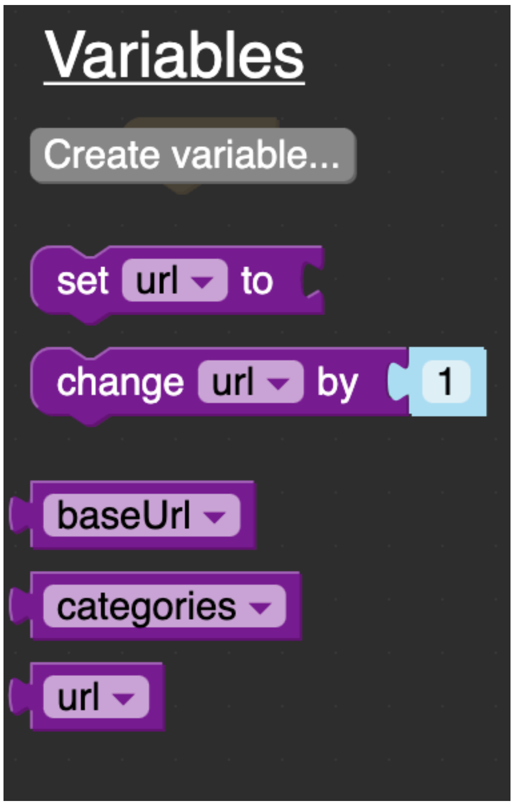
In our code we will create the URL we are going to call by adding some strings together, to make it easier for us to modify in the future.

Start off by creating some new Variables by clicking Variables / Create variable…

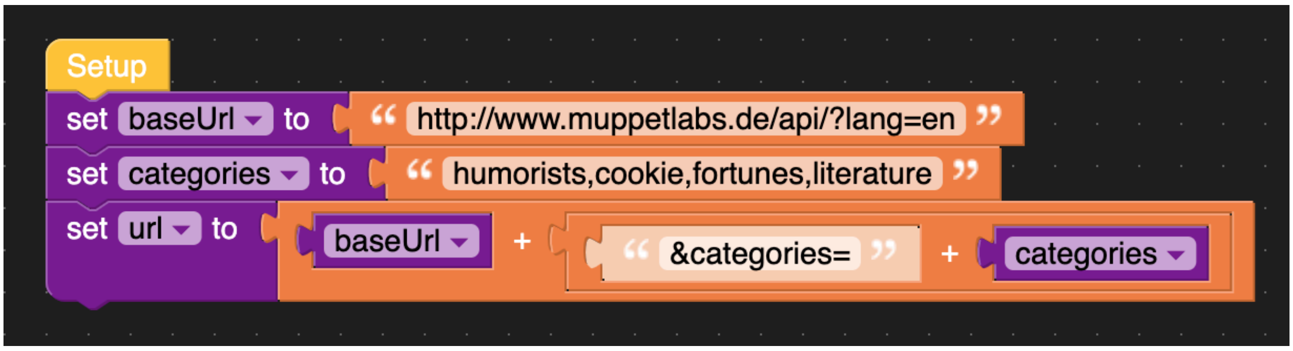
Create the following variables

[1] HTTP stands for Hypertext Transfer Protocol, and is the basis of how data is transferred from web servers to your web browser. See https://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol

```
* baseUrl
* categories
* url
```

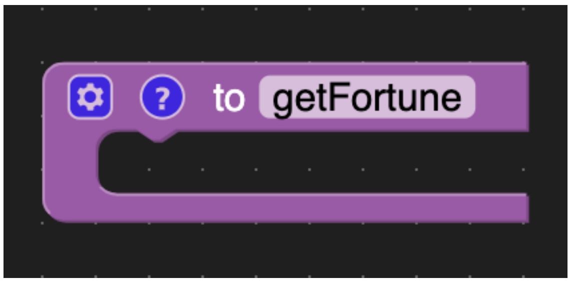


Now we can initialise the variables in our Setup section.



```
Setup
set baseUrl to "http://www.muppetlabs.de/api/?lang=en"
set categories to "humorists,cookie,fortunes,literature"
set url to baseUrl + "&categories=" + categories
```

Next we can make a start on the part of our code which fetches the fortune from the remote server. My preference is to use different functions to separate different concerns in my programs, so we will do that. Go to the Functions section, drag out the top block, and rename it `getFortune`.



Find the Advanced section and expand it, then click the Http section beneath it and drag out the Http Request block and drop it into your `getFortune` function.

Leave the **Method** set to `GET`.

An HTTP method is used to exchange data with a web server, usually when downloading a web page to display in your browser but in our case for downloading a fortune cookie.

HTTP requests are in plain text[2] so it is easy to follow what happens. The **client** sends a request to the **server**. The **request** consists of a few lines of plain text. The first line contains the request **method**, followed by the path being requested and the version of the HTTP protocol being used.

Subsequent request lines are called **Request Headers** and consist of keys and values to define various operating parameters for the request.

After the Request Headers are finished, the client finishes with a blank line, after which the server sends its **response**.

The response itself starts with a **status code**, followed by a number of header lines in the same format as the request headers. The actual response data follows after a blank line.

The HTTP status code you are probably most aware of is code **404**, which is sent by the server when the resource you requested could not be found.

GET /api/?lang=en&categories=humorists,cookie,fortunes,literature HTTP/1.1	Request line
Host: www.muppetlabs.de	Request header
Accept: */*	Request header
	⬅️ this is the blank line
HTTP/1.1 200 OK	First response header

Here is an actual HTTP request showing the request that the client sent (lines starting ">") followed by the response that the server sent (lines starting "<").

```
> GET /api/?lang=en&categories=humorists HTTP/1.1
> Host: www.muppetlabs.de
> User-Agent: curl/7.64.1
> Accept: */*
>
< HTTP/1.1 200 OK
< Content-Type: application/json
< Vary: Accept-Encoding
< X-Cloud-Trace-Context: 8a6d368377880996da215b7f03456a94
< Date: Mon, 15 Mar 2021 22:38:09 GMT
< Server: Google Frontend
< Content-Length: 137
<
{"fortune": "I never forget a face, but in your case I'll make an
exception.\n\t\t-- Groucho Marx", "source": ["fortunes/en/humorists"]}
```

[2] It gets a bit more complicated when you look at HTTPS requests (still plain text, but encrypted for transport) and HTTP version 2.0 and above.

All of this is just useful background information because most of the hard work will be done for us automatically by the Http Request block!

Drag the url variable to the URL field of the block.

Into the Success block, drag an Lcd.clear block followed by an Lcd.print block, and drag the “Get data” piece (from the Http section) into the string part.

At the end of the Setup block, drop in a function call for your getFortune function. Also drag out a “Button A was pressed” block from the Event section, change the button to C, add another call to getFortune in it.

Then send the program to your device.

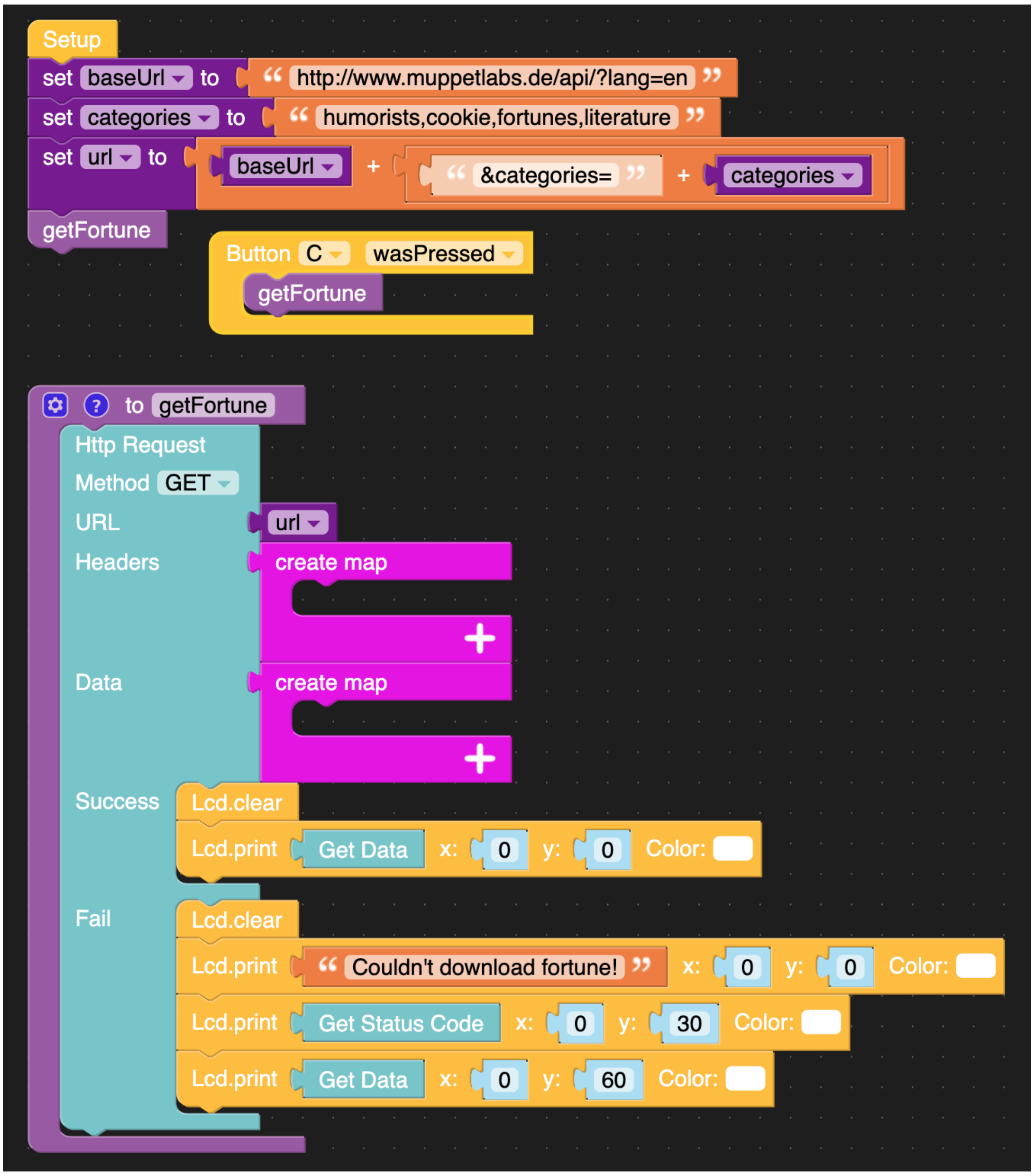
It should load a random fortune and put the raw unprocessed json[3] data on the screen. Every time you press button C, it will load another fortune.

Troubleshooting

To help diagnose any problems, add some code to the Fail section of the request.

- Lcd.clear
- Lcd.print “Couldn’t download fortune!” - at x:0 y:0
- Lcd.print Get Status Code - at x:0 y:30
    - You’ll find the Get Status Code block in the Http section
- Lcd.print Get Data - at x:0 y:60
    - You’ll find the Get Status Code block in the Http section

[3] JavaScript Object Notation



```
Setup
set baseUrl to "http://www.muppetlabs.de/api/?lang=en"
set categories to "humorists,cookie,fortunes,literature"
set url to baseUrl + "&categories=" + categories
getFortune

Button C wasPressed
    getFortune

to getFortune
    Http Request
    Method GET
    URL      url
    Headers  create map
                 +
    Data     create map
                 +
    Success  Lcd.clear
             Lcd.print Get Data x: 0 y: 0 Color:
    Fail     Lcd.clear
             Lcd.print "Couldn't download fortune!" x: 0 y: 0 Color:
             Lcd.print Get Status Code x: 0 y: 30 Color:
             Lcd.print Get Data x: 0 y: 60 Color:
```

Decoding json

The data in the server response is encoded in json, so we will have to decode it. The format (with newlines and indentation added to make it easier to read) is:

```
{
   "fortune": "I never forget a face, but in your case I'll make an
exception.\n\t\t-- Groucho Marx",
  "source": ["fortunes/en/humorists"]
}
```

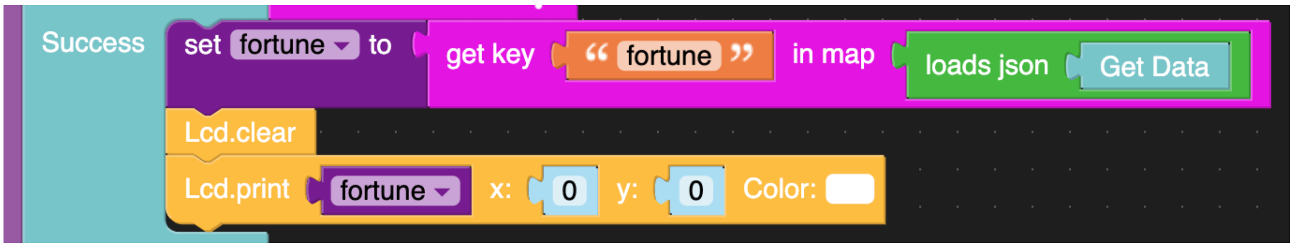
The curly braces surround an **object**, which has two **fields** - `fortune` and `source`. The `fortune` field is just a **string**, and the `source` field is an **array** of **strings**.

Contained in the string are some backslash characters followed by letters. These are called escape sequences. Here are some common escape sequences with their meanings.

\n	Newline
\t	Tab
\\	A literal backslash character "\"

For this program we only care about the fortune field so we will have to extract that. Fortunately there is a method we can use to decode the json data, and it will also handle the escape sequences for us.

Create a new variable called fortune, and into the Success section of your Http Request block, set up the following code:



```
Success
  set fortune to get key "fortune" in map loads json Get Data
  Lcd.clear
  Lcd.print fortune x: 0 y: 0 Color:
```

You will find the "loads json: Get Data" block at the top of the Http section in a subsection called Convert Json Data.

Notice that the Lcd.print block is now printing out the contents of your `fortune` variable.

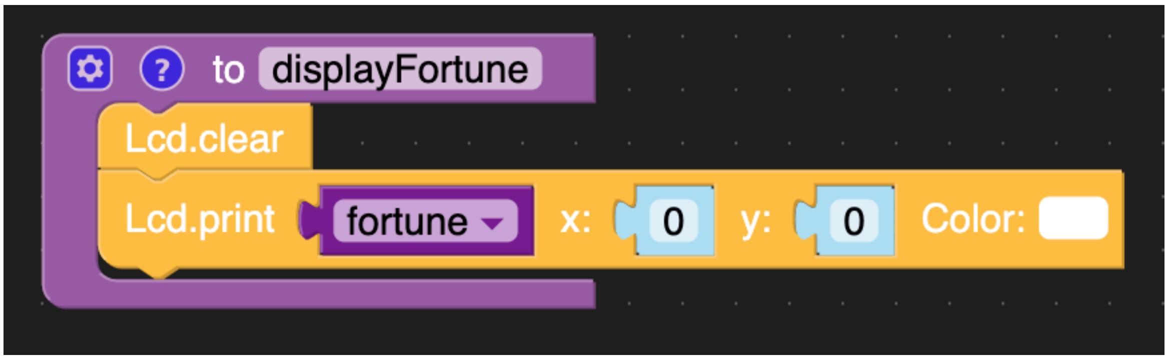
If you send this program to the M5Stack you should find that this time it looks a lot nicer because it just prints the fortune.

Auto-load

We are going to enhance our program slightly now by adding the following features:

- Auto-load a new fortune every 30 seconds
- Flash the light when a new fortune is loaded.

We will start by creating a new displayFortune method and moving part of our existing Success code to the new function.



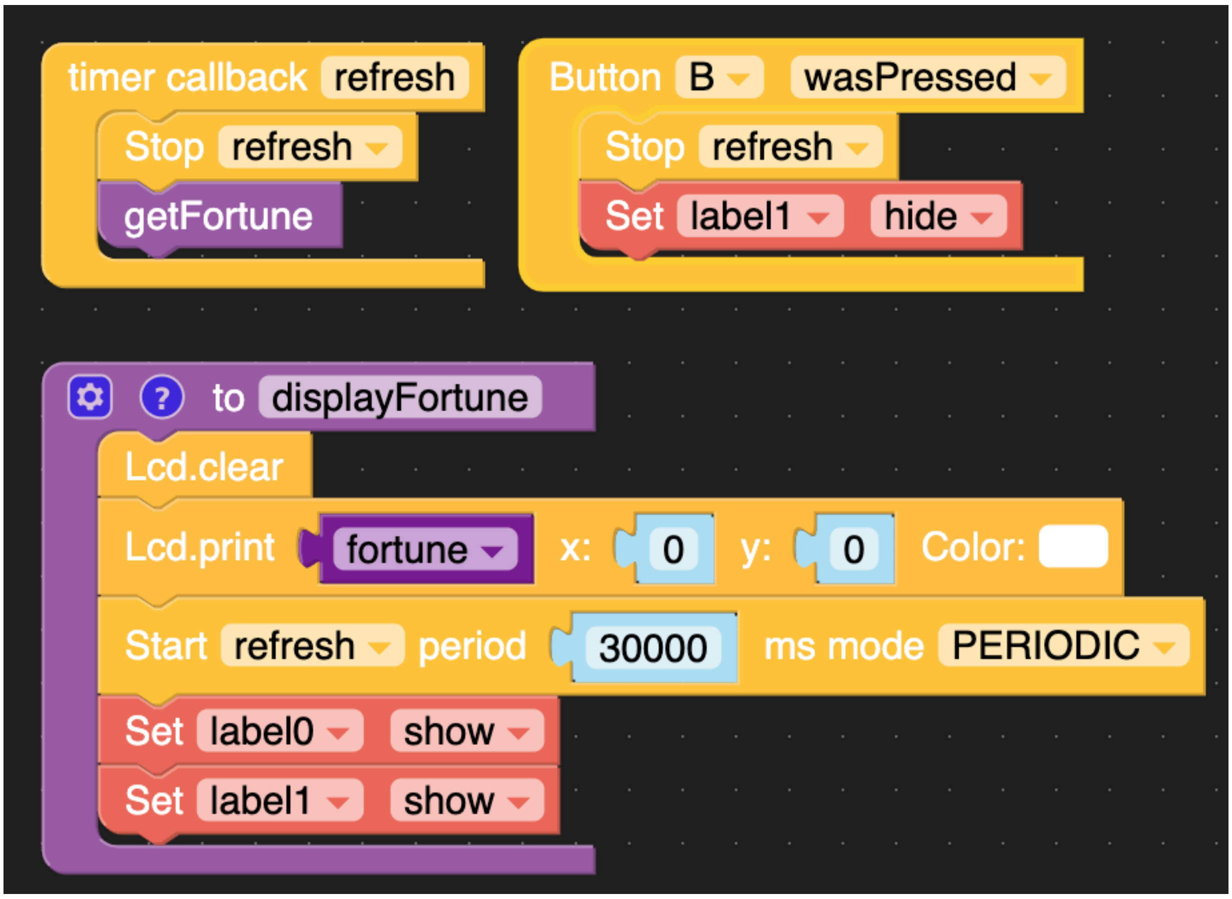
```
to displayFortune
  Lcd.clear
  Lcd.print fortune x: 0 y: 0 Color:
```

Drag out a new Timer Callback block and call it refresh. Into that block, put a call to your getFortune function. At the end of your displayFortune block, make it start a 30 second timer to refresh the screen. Add a “Stop refresh” block to the top of the refresh timer callback. There is a ONE_SHOT mode for the timer which would be ideal, but it doesn’t seem to work properly!

In your getFortune function, drag out a “Set RGB Bar color: green” to the top (just before the Http Request), and a “Set RGB Bar color: black” to the end (just after the Http request).

On the picture of your device, drag a new label and position it at the bottom of the screen above button C, and change the text to “Another”, to label the fact that pressing Button C will load another fortune. You might also like to pick a nice colour for it. Add “Set label0 show” to the end of displayFortune (because the Lcd.clear will have removed it). Because adding a label to the screen will automatically set the screen font unless you specify otherwise, you could also add a Font: `FONT_DejaVu18` block to the top of your Setup section.

I also found it useful to add another label “Pause” above button B, and add some code to cancel the timer. You’ll also need to add a “show” block for that to the end of displayFortune.



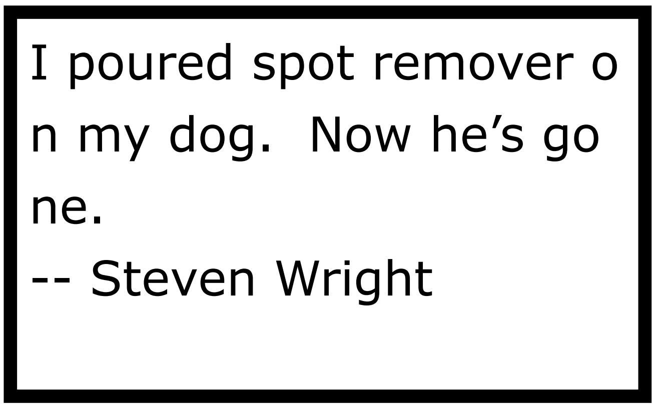
```
timer callback refresh
    Stop refresh
    getFortune

Button B wasPressed
    Stop refresh
    Set label1 hide

to displayFortune
    Lcd.clear
    Lcd.print fortune x: 0 y: 0 Color:
    Start refresh period 30000 ms mode PERIODIC
    Set label0 show
    Set label1 show
```

Word wrapping algorithm

At this point you should have a program that basically works, and it might be that this is all we have time for in the session. However you will probably have noticed one of the problems with the display, which is that the words get broken at the end of lines. This section is quite advanced!



```
I poured spot remover o
n my dog.  Now he’s go
ne.
-- Steven Wright
```

If you print text to the screen with Lcd.print, it will wrap the text onto the next line as soon as the previous line is filled, even if it’s in the middle of a word. This doesn’t look very good so we will attempt to create our own word wrapping algorithm.

The general form of the algorithm will be:

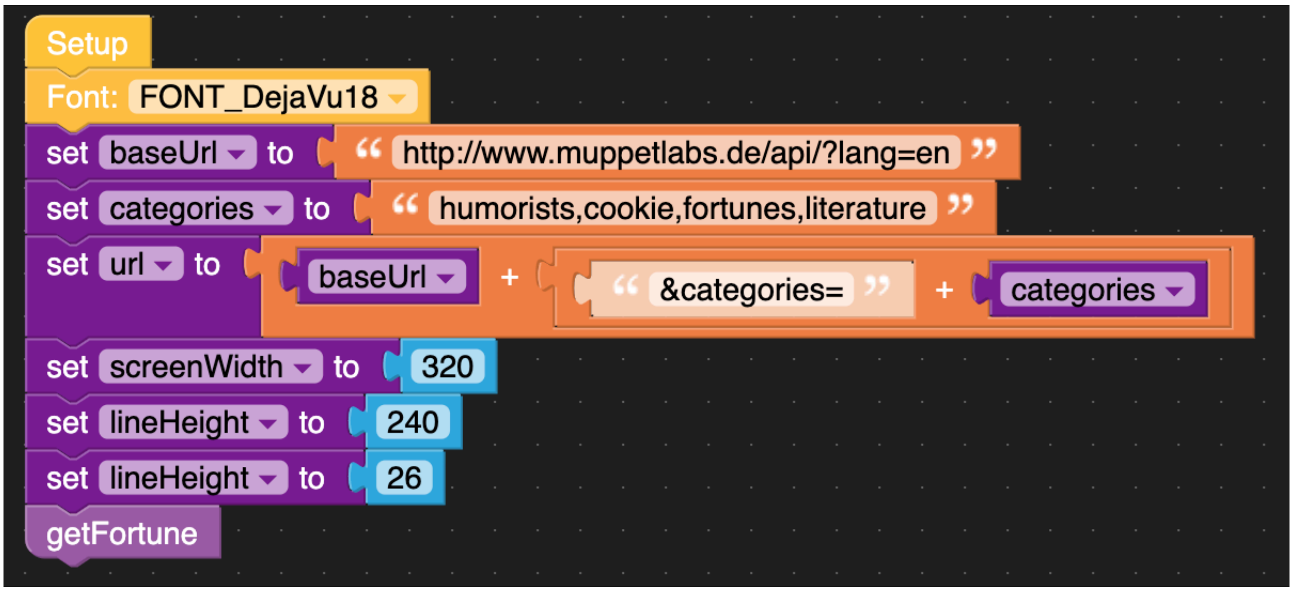
1. Split the text into individual words
2. Always remember the x and y coordinates where we are about to draw a word
3. For each word, work out how many pixels wide it is before displaying it
4. If it fits on the current line then draw it
5. Increment the current x position by the width of the word + the width of a space character
6. If it doesn’t fit on the current line then increase the y position by the line height, set the x position back to 0, and draw the word there (at the start of the next line)
7. Look out for a line that starts with the “--“ characters and make sure we always move onto a new line for those because it’s usually the “attribution” line.

We will start by adding a number of variables to track things like screen dimensions, current x and y position, line height etc.

Create new variables with the following names

- `screenWidth`
- `screenHeight`
- `lineHeight`
- `lineNumber`
- `currentLineY`
- `currentWordX`
- `fortuneLines`
- `fortuneWords`
- `wordWidth`

Add values for the first three of these to your Setup block. The screen dimensions we know from the specification of the device, and the lineHeight is the number of pixels from the top of one row of text to the top of the next row - I worked it out by trial and error but it will be different for each font.



```
Setup
Font: FONT_DejaVu18
set baseUrl to "http://www.muppetlabs.de/api/?lang=en"
set categories to "humorists,cookie,fortunes,literature"
set url to baseUrl + "&categories=" + categories
set screenWidth to 320
set lineHeight to 240
set lineHeight to 26
getFortune
```

At the top of your displayFortune method, just after Lcd.clear, set `lineNumber` and `currentWordX` to `0`.

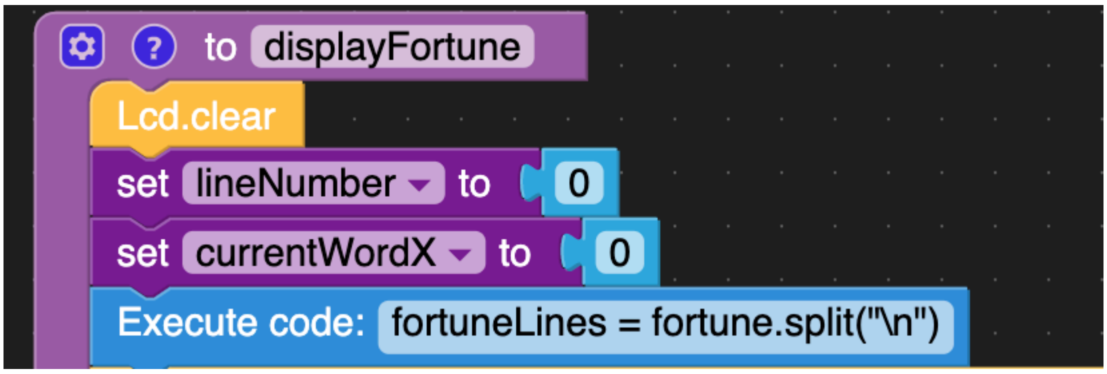
Now we are ready to split the text. First we will split it on each newline character, so we get an array of strings, one per line. Each line will be further processed by splitting on each space character.

The python method we want to use is called split() - you call it on a string, it has a single argument (the character you want to split on), and it returns an array of strings.

Unfortunately, it isn't directly available in UI Flow, but there is a block we can use - the **Execute code** block from the Advanced / Execute section. This block allows us to type in the python code we want to execute, without having to abandon the Blockly environment.

Add an Execute code block containing the following code:

```
fortuneLines = fortune.split("\n")
```



```
to displayFortune
  Lcd.clear
  set lineNumber to 0
  set currentWordX to 0
  Execute code: fortuneLines = fortune.split("\n")
```

This will split the contents of the existing fortune variable into a List of strings, one per line, and store that List of strings in the fortuneLines variable. "\n" is the escape sequence for a newline character.

We want to run some code for each item in the List, and in programming terminology that's known as iterating over the List.

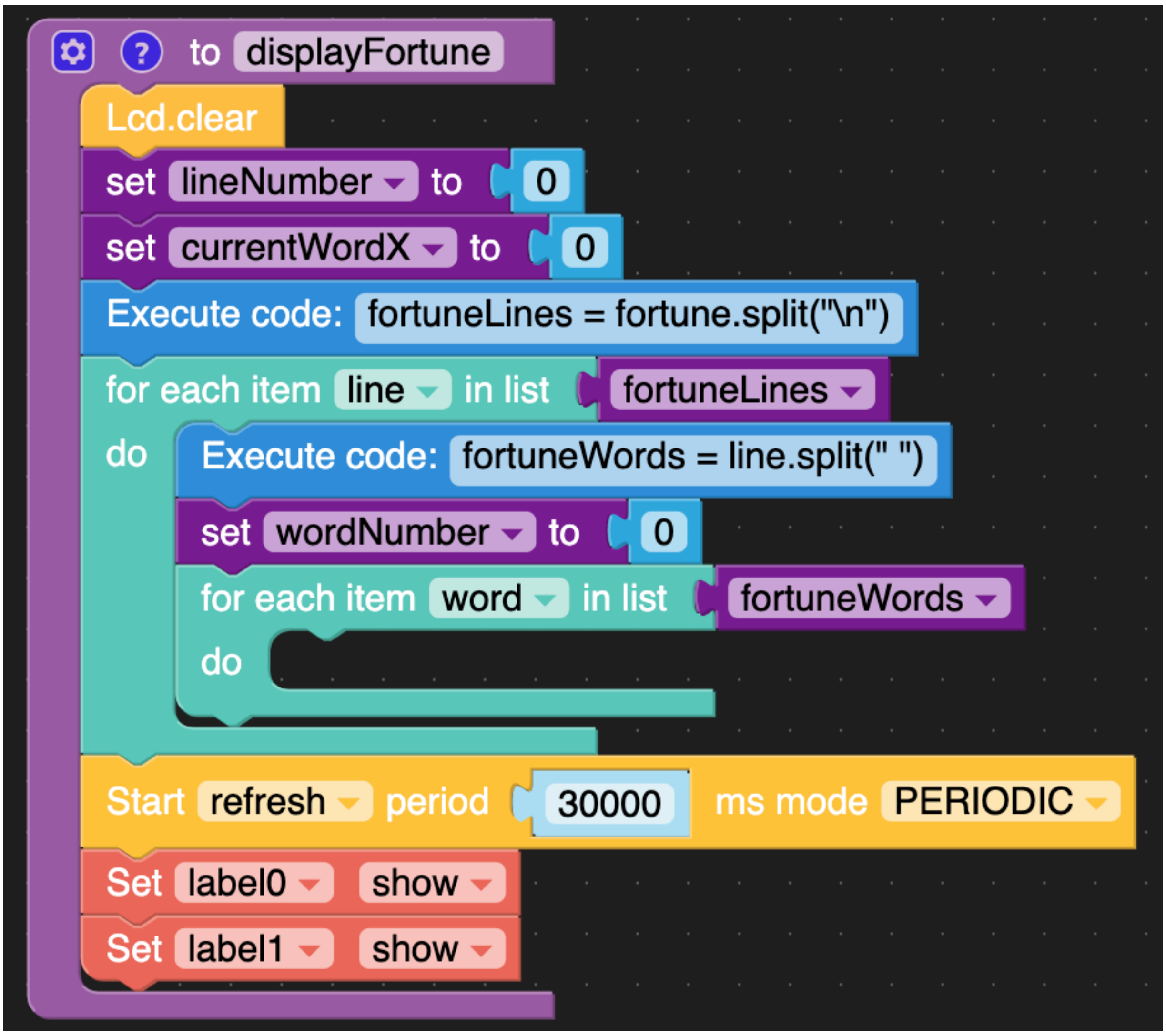
In the Loops section, drag out the block "For each item i in list", rename the variable "i" to "line", and drag out the fortuneLines variable and attach it to the iterator block.

Each time through the loop ("each iteration"), the `line` variable will be set to the next line in the List. We want to split that line into words. We want to keep track of how many words we've processed on the current line, so we also reset the `wordNumber` variable to `0` at the top of the loop.

We split the line into words with another Execute code block containing this code:

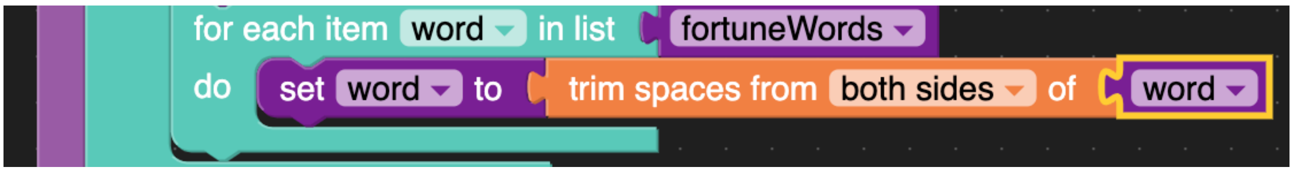
```
fortuneWords = line.split(" ")
```

After that, we can add another iterator block to iterate over the words in the `fortuneWords` list. Rename the iterator variable from "i" to "word".



```
to displayFortune
    Lcd.clear
    set lineNumber to 0
    set currentWordX to 0
    Execute code: fortuneLines = fortune.split("\n")
    for each item line in list fortuneLines
    do
        Execute code: fortuneWords = line.split(" ")
        set wordNumber to 0
        for each item word in list fortuneWords
        do
    Start refresh period 30000 ms mode PERIODIC
    Set label0 show
    Set label1 show
```

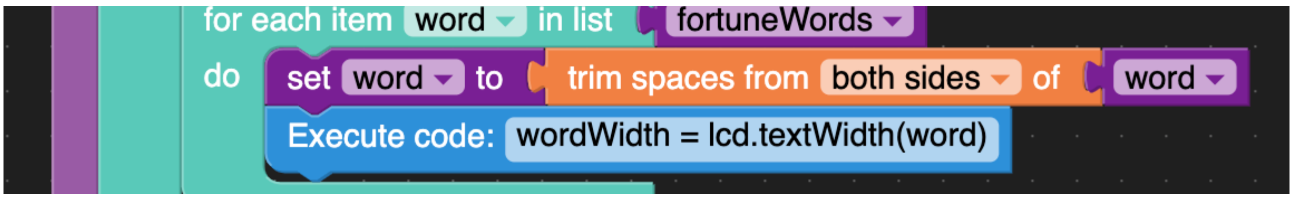
Inside the loop, the variable `word` will be set to each word in the list in order, changing every time through the loop. The first thing we are going to do is clean up the contents of the word variable to get rid of any tab characters that there might be (remember the "`\t`" from the raw json?). We can do that with the "trim spaces" block.



```
for each item word in list fortuneWords
do  set word to trim spaces from both sides of word
```

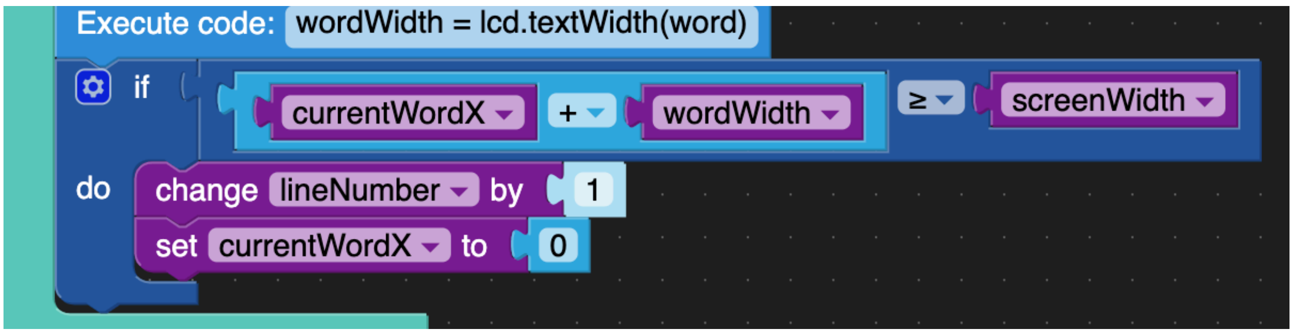
To work out if a word is going to fit on the current line, we need to know how wide it is. There is a method available to do this, but we will need to use another execute block to be able to call it. The code to execute is:

```
wordWidth = lcd.textWidth(word)
```



```
for each item word in list fortuneWords
do  set word to trim spaces from both sides of word
    Execute code: wordWidth = lcd.textWidth(word)
```

We need some code to check if that many (wordWidth) pixels will fit on the current line. currentWordX is the x position where we are about to draw the word, wordWidth is the width of the word, and screenWidth is the width of the screen in pixels. Add an If / Do block with the following test:



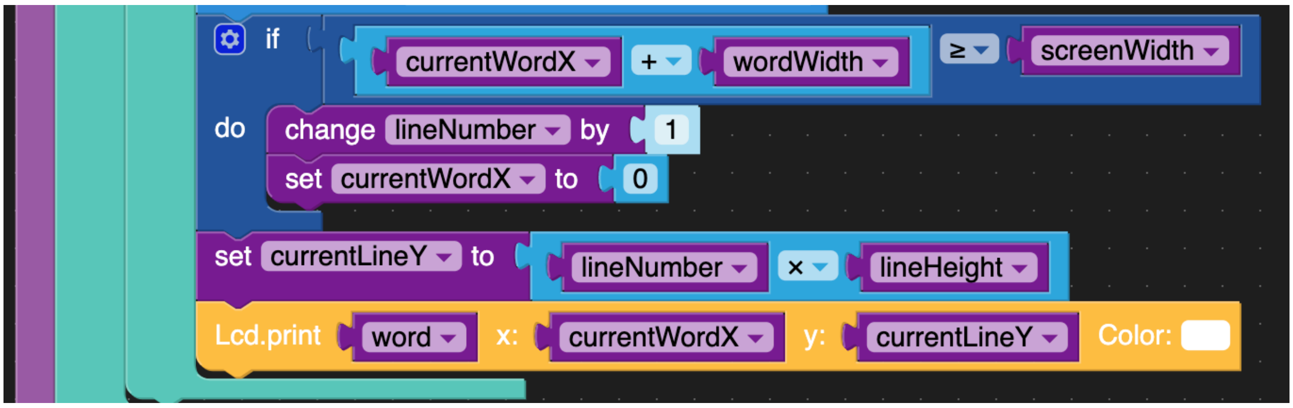
```
Execute code: wordWidth = lcd.textWidth(word)
if  currentWordX + wordWidth ≥ screenWidth
do  change lineNumber by 1
    set currentWordX to 0
```

So the logic is:

If the current word X position plus the width of the word we want to write is greater than or equal to the screen width, then increment the line number and set the word X position back to 0 (so we will write the word at the start of the next line).

If it fits then do nothing special because we are about to draw it in the correct position.

First we need to work out the y position of the text, which is simply the `lineNumber` times the `lineHeight`.

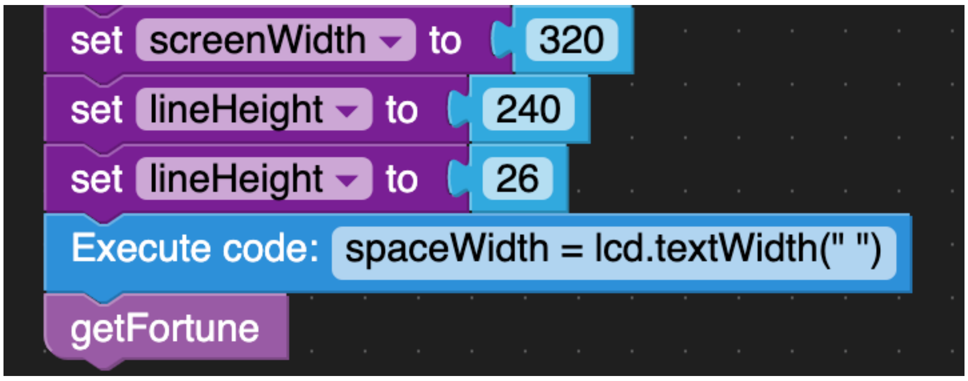


```
if  currentWordX + wordWidth ≥ screenWidth
do  change lineNumber by 1
    set currentWordX to 0
set currentLineY to  lineNumber × lineHeight
Lcd.print  word  x: currentWordX  y: currentLineY  Color:
```

Now that we know the X and Y positions for the word, we can draw it with Lcd.print.

We need to increment `currentWordX` by the width of the word we just printed - plus the width of a space character. We will have to add an Execute Code block to our Setup block containing this code:

```
spaceWidth = lcd.textWidth(" ")
```



```
set screenWidth to 320
set lineHeight to 240
set lineHeight to 26
Execute code: spaceWidth = lcd.textWidth(" ")
getFortune
```

Also make sure you create a variable called spaceWidth.

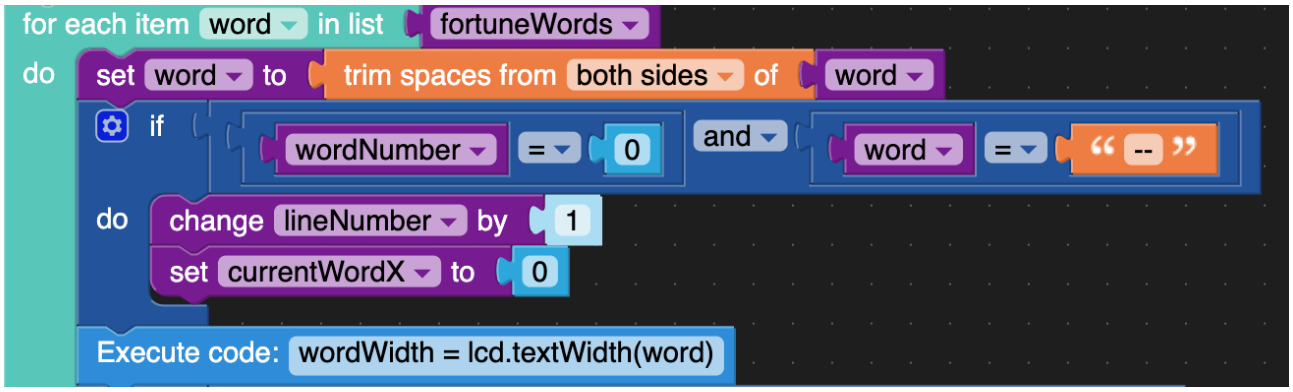
Our final enhancement will be to make it put the attribution on its own line at the end. Our logic will be:

If the currentWordNumber is 0 (the first word on the line)

AND if the current word is "--" (two hyphens)

THEN increment the line number and set the x position back to 0

We need to insert this code right near the top, just after we trim the spaces from either side of the word.



```
for each item word in list fortuneWords
do  set word to trim spaces from both sides of word
    if  wordNumber = 0  and  word = " -- "
    do  change lineNumber by 1
        set currentWordX to 0
    Execute code: wordWidth = lcd.textWidth(word)
```

Bugs / enhancements

1. Although it was one of the reasons for this demo, there is no actual reason for this program to use an internet connection. The fortune database files could be installed on the device
2. Sometimes the fortune is too long for the screen - it would be great if it could scroll, or just skip that fortune and fetch another
3. Some of the fortunes end with an attribution, e.g. "`-- Rodney Dangerfield`". Maybe display that in a different font or colour.
4. It could remember each fortune it had displayed, so we could change the button functions to Previous / Pause / Next.