

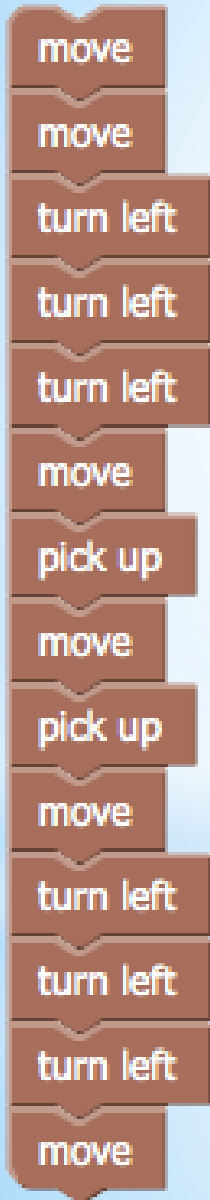
Programming 1 / Object Oriented Programming

Thinking Like a Programmer
Lecture #2: The fundamental
programming constructs

Programming is like lego

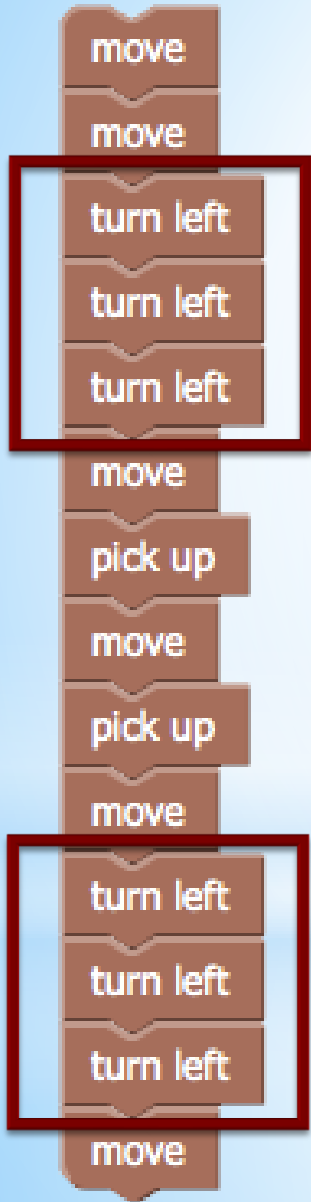
- ⦿ Regardless of what programming language you're using, there will (usually) be the same fundamental constructs:
 - ⦿ Conditional statements (making decisions)
 - ⦿ Repetition
 - ⦿ Conditional repetition
 - ⦿ Defining new functions
- ⦿ Programming is like building something with lego - the fundamental constructs are the bricks
- ⦿ Some bricks fit together - some do not!
- ⦿ It is up to you to assemble the bricks in an order that makes sense to build

The story so far...



- ⦿ You probably ended up with solutions that looked a lot like this in the latter part of last week's workshop...
- ⦿ "Carol is stupid" - this we know; she only has a very limited set of commands
- ⦿ We've had to have a lot of repetition in our programs so far to compensate for the lack of commands
- ⦿ Wouldn't it be great if we could create *new* commands to address Carol's limitations and reduce this repetition of code?
- ⦿ Can anyone think of a new command for Carol that'd be really handy to have?

The story so far...



- ⦿ Carol has no single command to turn right
- ⦿ We've had to turn left three times to simulate turning right
- ⦿ This involves a lot of repetition whenever we want to turn right
- ⦿ Instead, we can use a *function* to tell Carol how we turn right
- ⦿ Once we've *defined* this turn right function, we can *call* it as many times as we like

Functions

- ⦿ A function is a section of a program that defines how to perform a specific task (e.g. "turn right")
- ⦿ A function has a name
- ⦿ The function contains the instructions required to perform the task
- ⦿ In some programming languages, a function might be called a *procedure* or a *method* - but the principle is the same
 - ⦿ ...we'll use *function* as the terminology in our Carol blocks...

Using the Carol function blocks

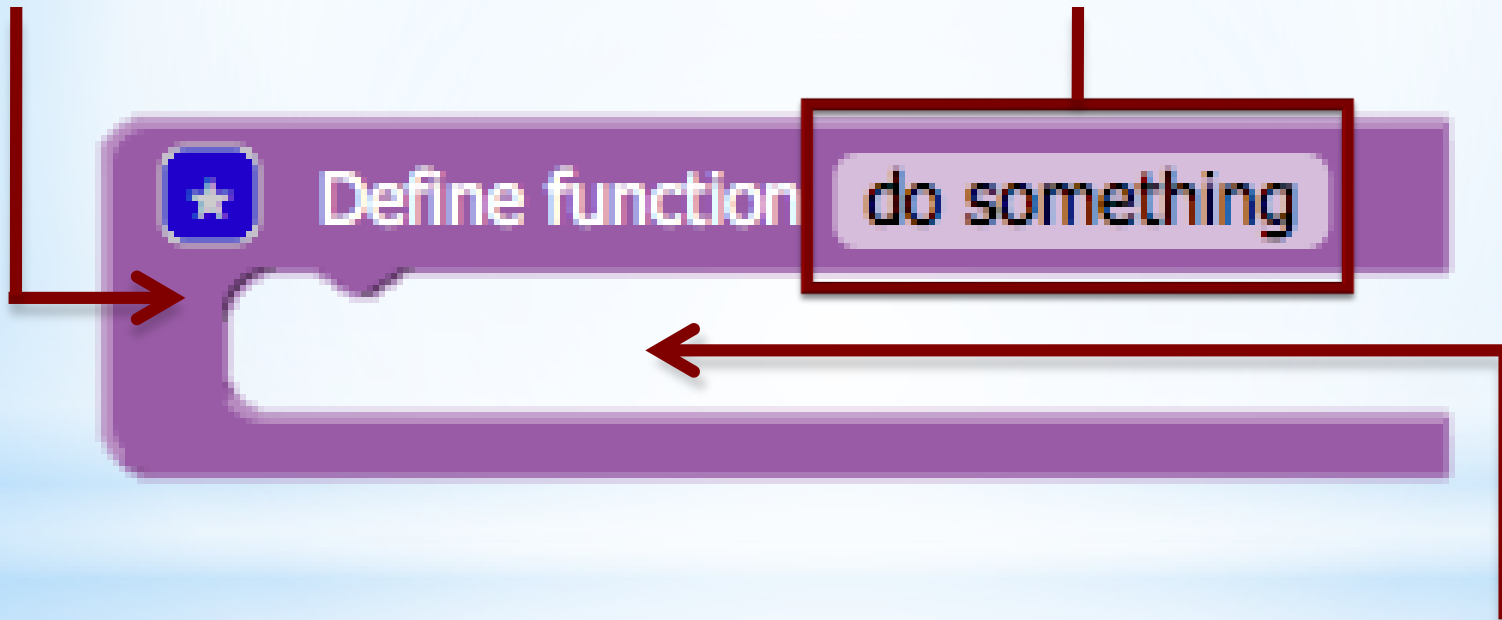
- ⦿ You can define a function using the blocks underneath the *Functions* menu

The screenshot displays the Carol programming interface. On the left is a vertical menu with the following categories: Display, Text, Variables, Decisions and loops, Comparisons, Numbers and maths, Functions (highlighted in blue), and Carol. The main workspace on the right contains three purple function blocks. The top two blocks are 'Define function' blocks, each with a star icon, the text 'do something', and a large empty space for code. The bottom block is an 'if' block with a 'return' block nested inside it. The 'if' block has two empty slots for conditions, and the 'return' block has two empty slots for return values.

Using the Carol function blocks

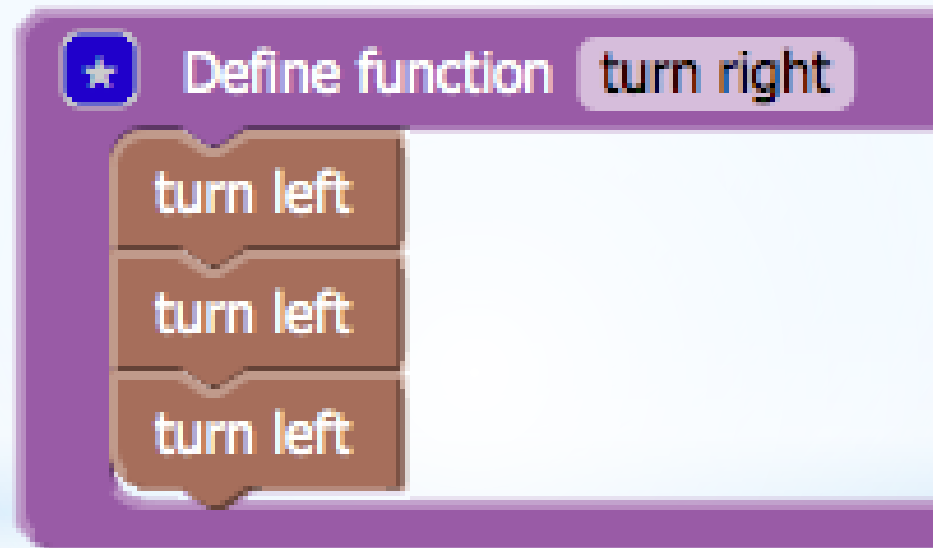
#1: Drag your function block onto your program canvas area*

#2: Click here and type to give your function a name



#3: Drag the instructions that perform the task of your function into this gap

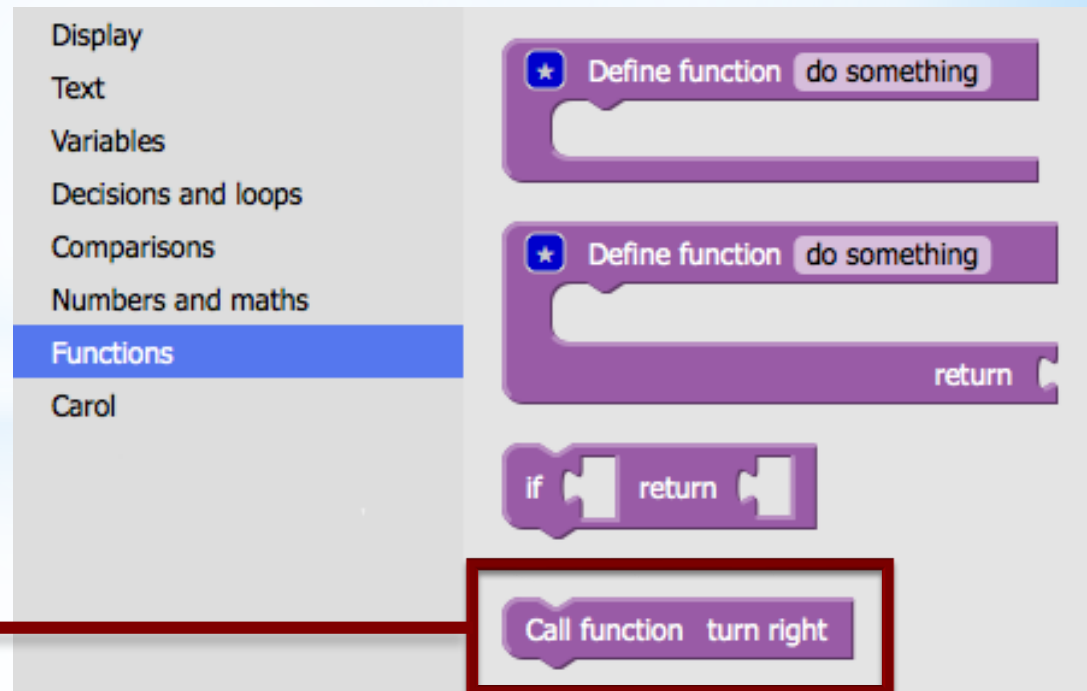
An example "turn right" function



Using our "turn right" function

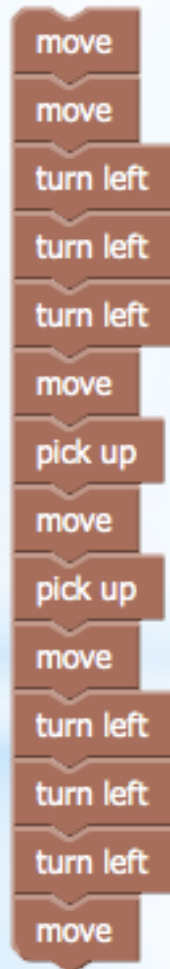
- ◉ Whenever you declare a function, an extra block will be added underneath the *Functions* menu item
- ◉ You can drag this block into your program to call your function

This block only appears if we declare a function called *turn right*

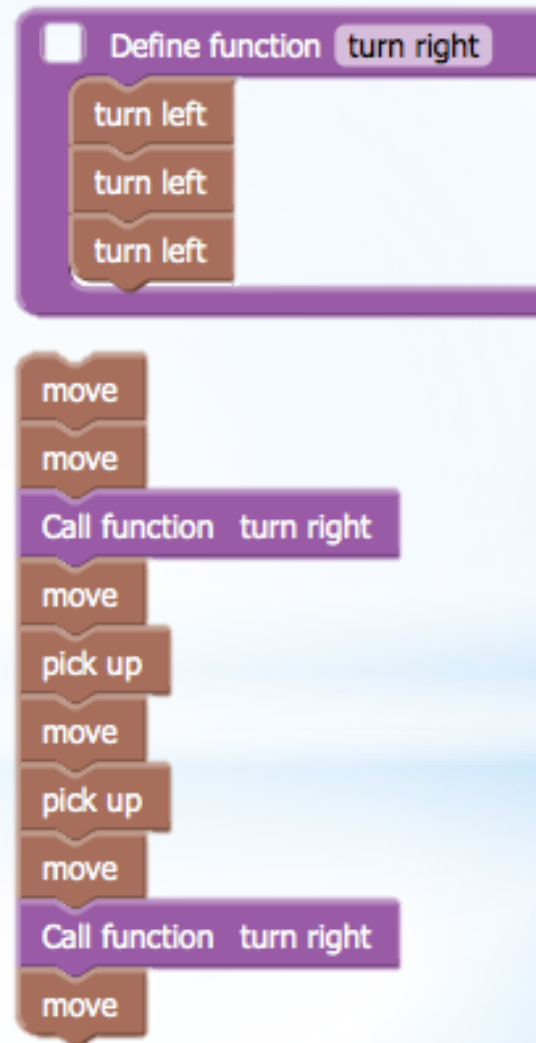


Using our "turn right" function

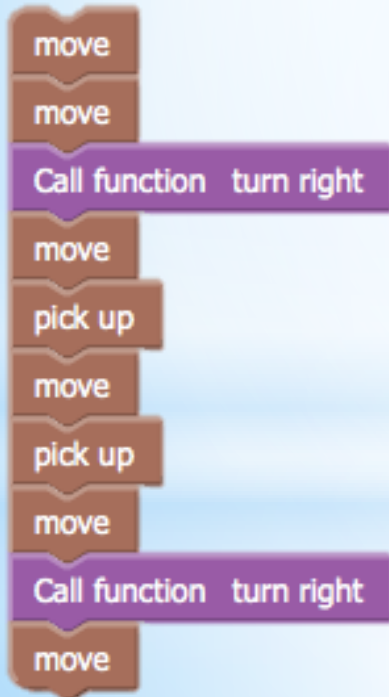
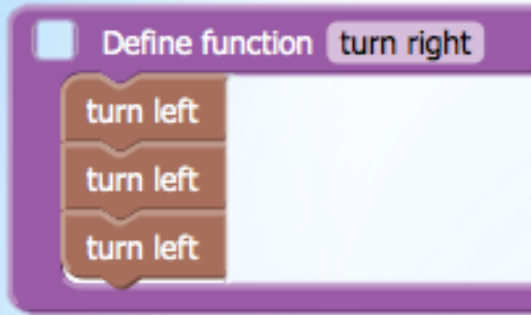
Before



After

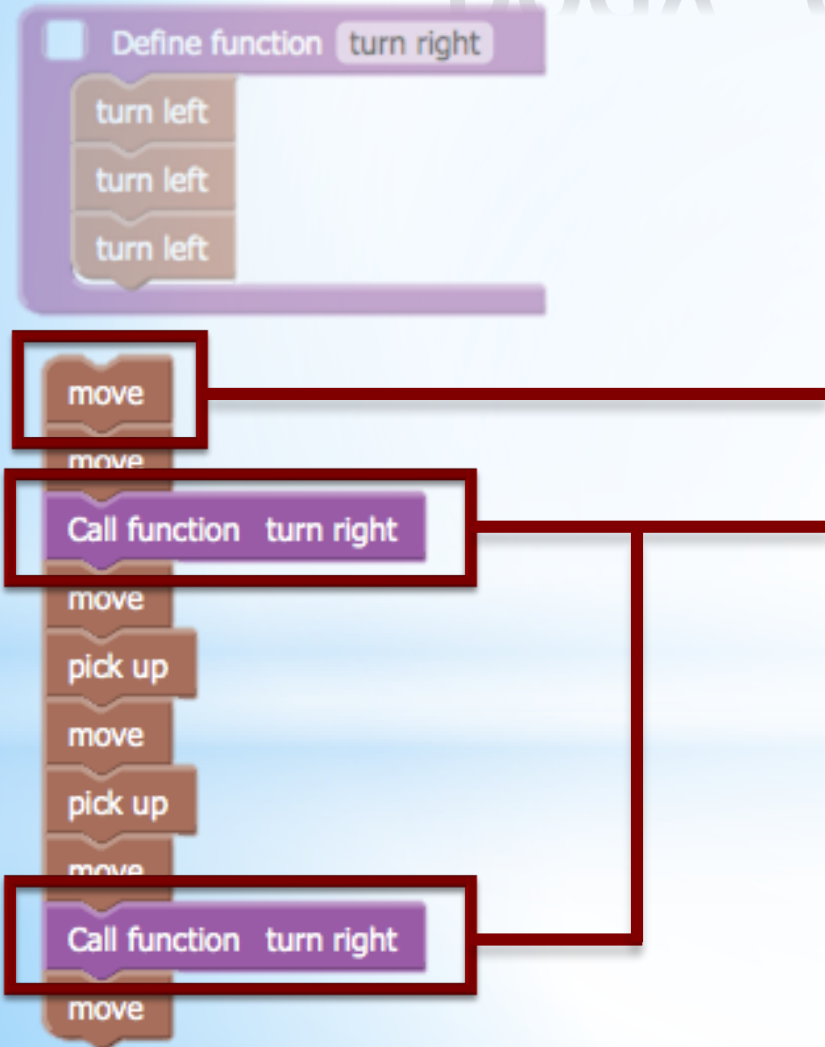


Using our "turn right" function



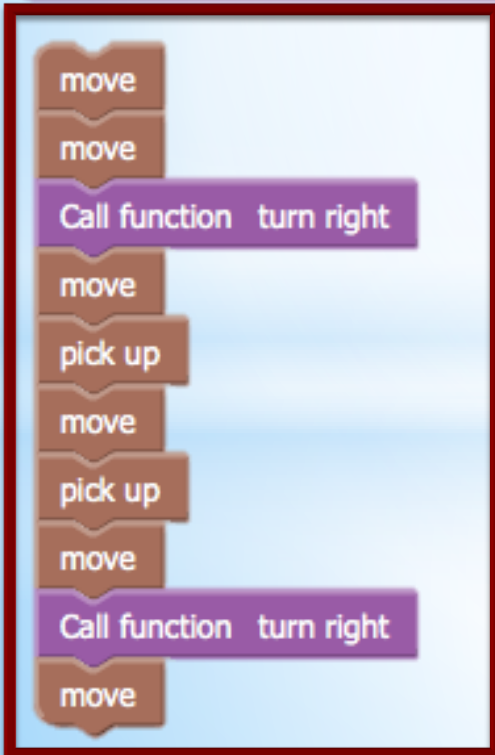
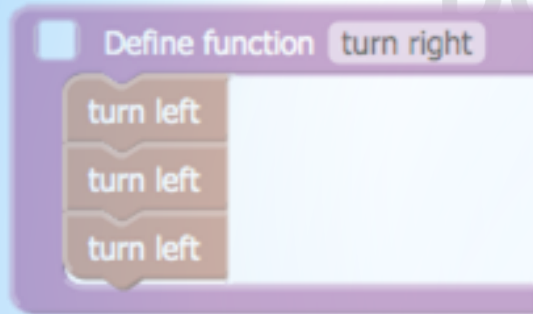
- ⦿ When you declare a function, you are simply explaining how to do something new
- ⦿ The instructions in the function **ONLY** happen when the function is called
- ⦿ So, with that in mind, what is the first line of this program that will run?

Functions and the "main body" of your programs



- This is the first instruction of the program to run
- Even though the function appears as the first thing in the program, the code inside it does not get run until it is called
- So, if a piece of code is inside a function, think of it as being "saved for later on"
- **THIS IS A CRUCIAL CONCEPT TO GRASP!**

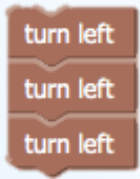
Functions and the "main body" of your programs



The part of your program outside of the functions is called the *main body* of your program

The program will ALWAYS start running at the first line of the main body

When to use functions - some guidelines and thoughts

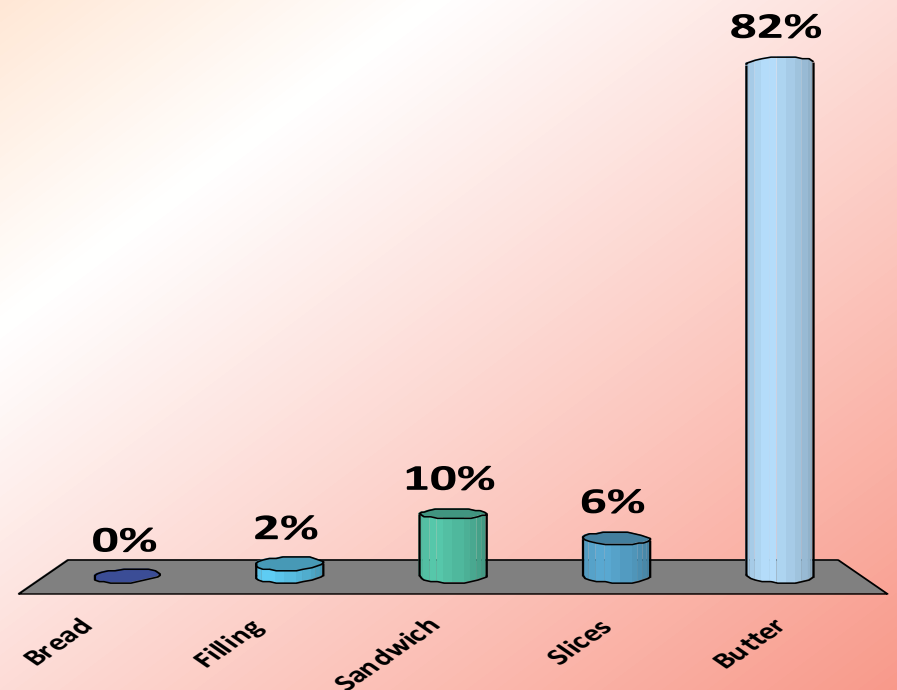
- ⦿ If you find yourself using the same sequence of instructions repeatedly within the same program, you should probably put them inside a function
- ⦿ e.g.  strewn throughout your program!
- ⦿ As you become more skilled in the art of thinking like a programmer, you will be able to spot things that might need to be functions up front before you write the code
- ⦿ By all means write these functions in advance...
 - ⦿ For example, if you spotted that a particular Carol grid will often need you to move a distance four squares, perhaps you should write a "move 4 squares" function before anything else...
- ⦿ It might help to think of functions as a way of creating brand new commands (e.g. "turn right")

Making a sandwich:

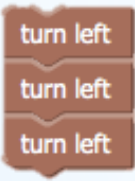

First, get the bread. Then, get the filling. Butter both sides of the bread. Put the filling between the two slices. Then enjoy your sandwich!

In our sandwich “program” from last week, what concept might be a good idea for a function?

- A. Bread
- B. Filling
- C. Sandwich
- D. Slices
- E. Butter



Repetition

- One of the reasons why computers are powerful tools is that they're very good at doing the same thing repeatedly - they do not get bored or tired
- We've already made programs with repeated instructions, e.g.  and 
- It's a bit of a pain in the backside to have to keep dragging the same block onto the canvas...
- What if we had a 20 x 20 grid and wanted to move from one side to the other?
- What if we had a 200 x 200 grid?!!!

The for loop

- ⦿ The for loop is one of the basic constructs of programming
- ⦿ Every programming language has some variation of it
- ⦿ You specify
 - ⦿ A start number
 - ⦿ An end number (or condition)
 - ⦿ A counter variable
 - ⦿ we'll talk more about *variables* later on
 - ⦿ (optionally) What happens to the counter variable each time the loop repeats

for loops in Carol blocks

Display

Text

Variables

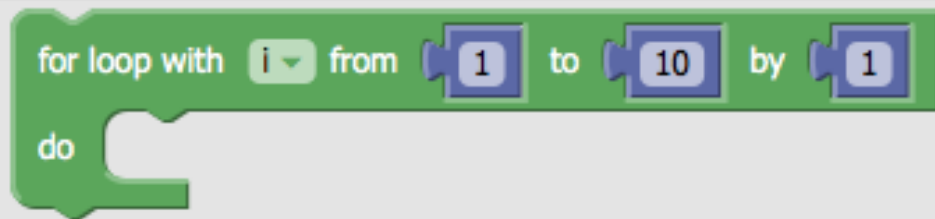
Decisions and loops

Comparisons

Numbers and maths

Functions

Carol

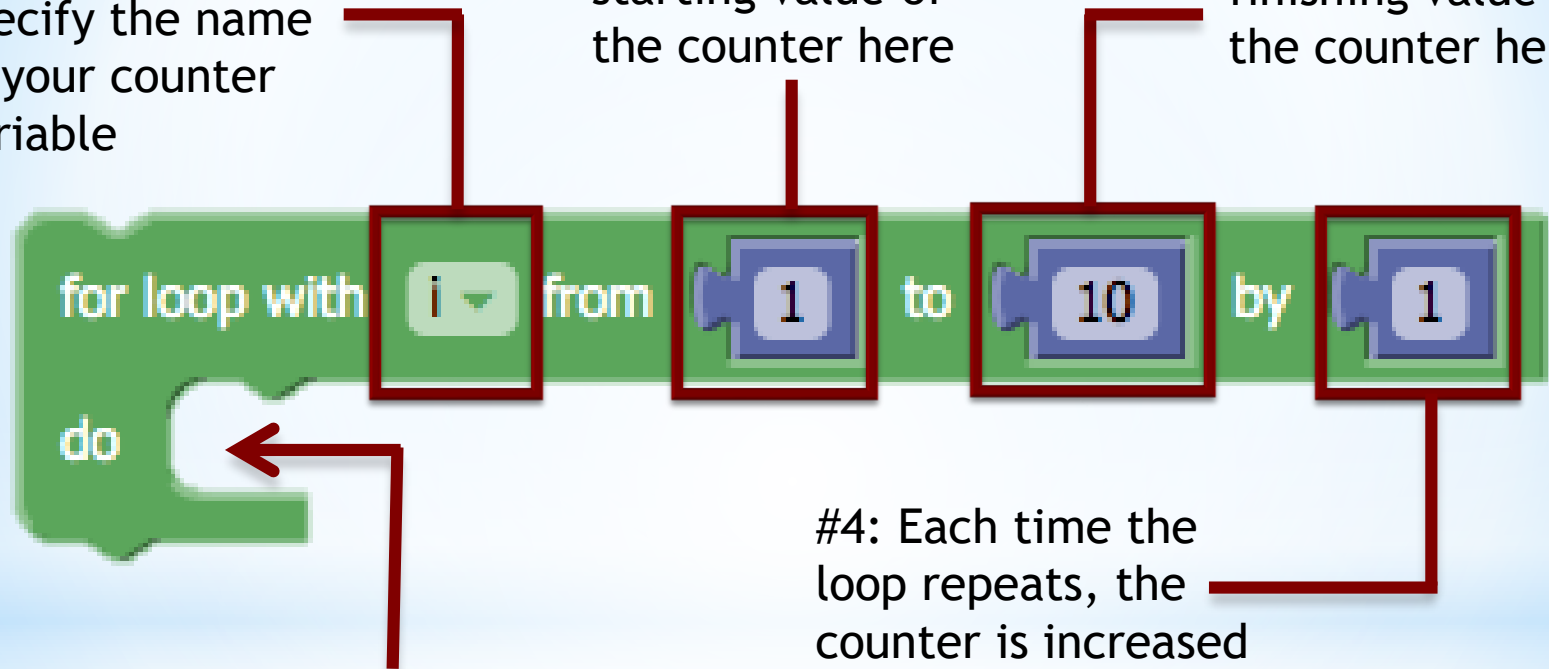


for loops in Carol blocks

#1: Click here to specify the name of your counter variable

#2: Specify the starting value of the counter here

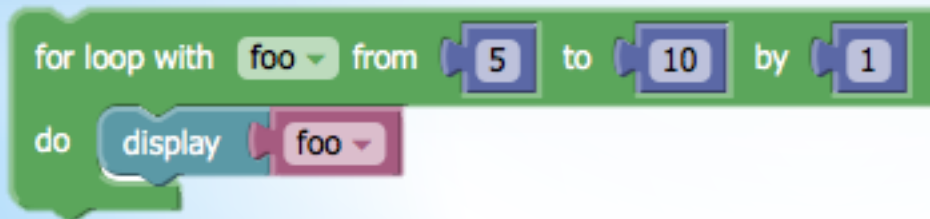
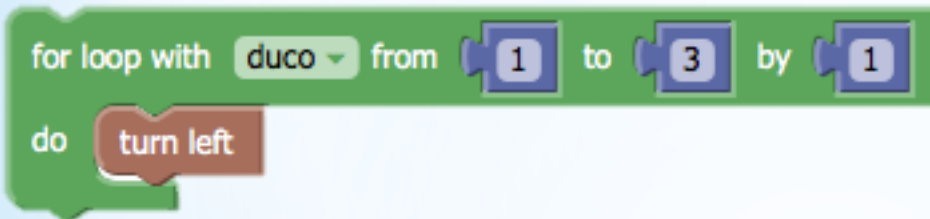
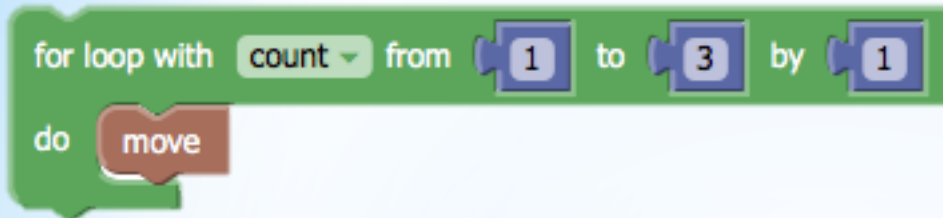
#3: Specify the finishing value of the counter here



#5: Drag the instructions you want repeated into this gap here. There can be as many as you like

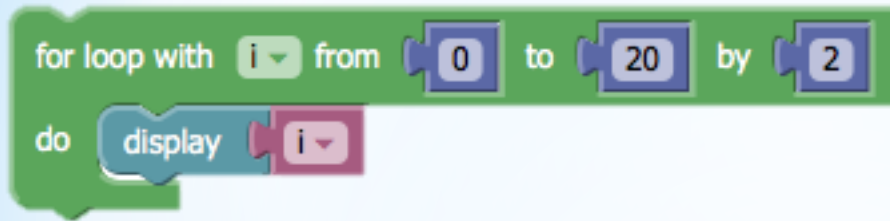
#4: Each time the loop repeats, the counter is increased by this value

for loop examples



NB: the *display* block displays text or a number to the screen

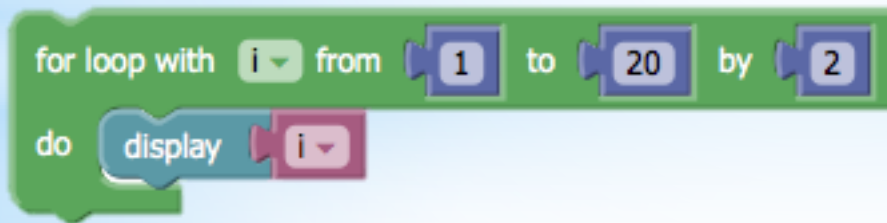
for loop examples



Displays

0
2
4
6
8
10
12
14
16
18
20

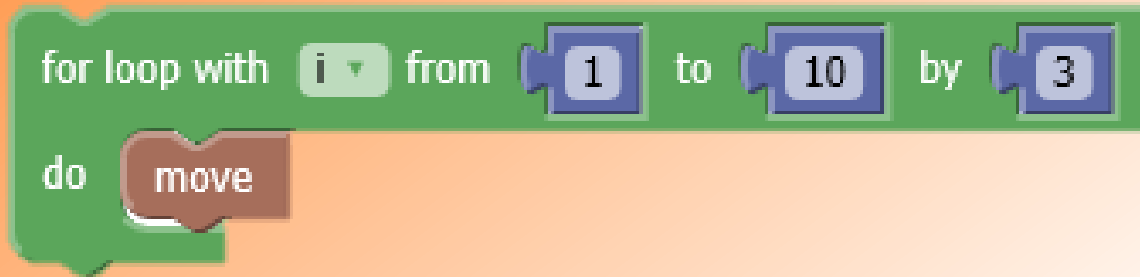
WHY?



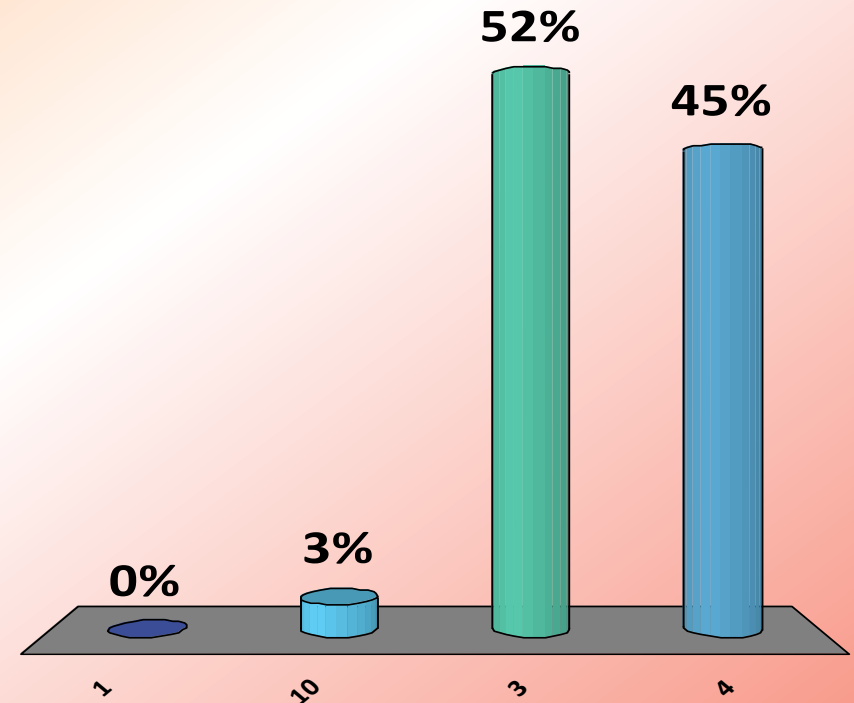
Displays

1
3
5
7
9
11
13
15
17
19

How many times would Carol move given the blocks below?

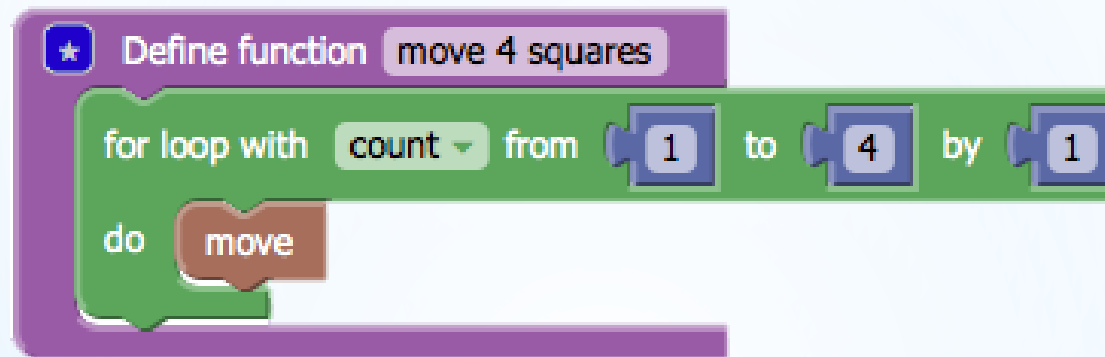


- A. 1
- B. 10
- C. 3
- D. 4

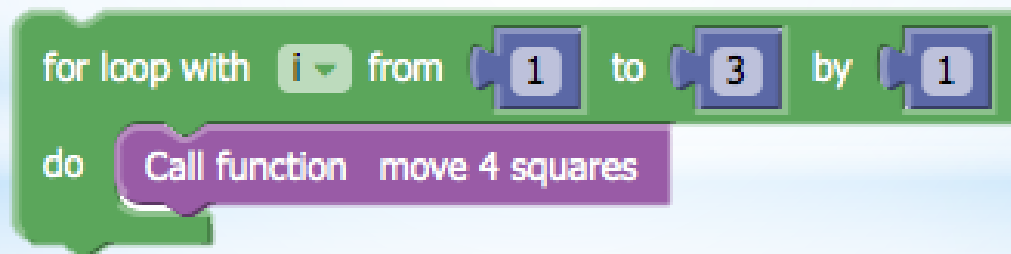


If the block fits, wear it...

- ⦿ You can put blocks within blocks, if you need to...
- ⦿ For example, a for loop can go within a function...



- ⦿ ...and a for loop can *call* a function...



- ⦿ (although a for loop can't actually contain a function definition... why do you think this is?)

Conditional processing (the *if* statement)

- ⦿ All of our Carol mazes so far have been static
- ⦿ What about if we wanted to write a program for a maze that had variable configurations?
- ⦿ What if we needed to check if the path ahead was blocked and take different action if the coast was clear?

The *if* block

The image shows the Scratch block palette on the left and the code area on the right. The palette categories are: Display, Text, Variables, Decisions and loops (highlighted in blue), Comparisons, Numbers and maths, Functions, and Carol. In the 'Decisions and loops' category, the 'if' block is highlighted with a red square. The code area contains three green loop blocks: a 'for loop with i from 1 to 10 by 1' block, a 'while do' block, and a 'repeat until' block.

Display
Text
Variables
Decisions and loops
Comparisons
Numbers and maths
Functions
Carol

if
do

for loop with *i* from 1 to 10 by 1
do

while
do

repeat
until

The *if* block

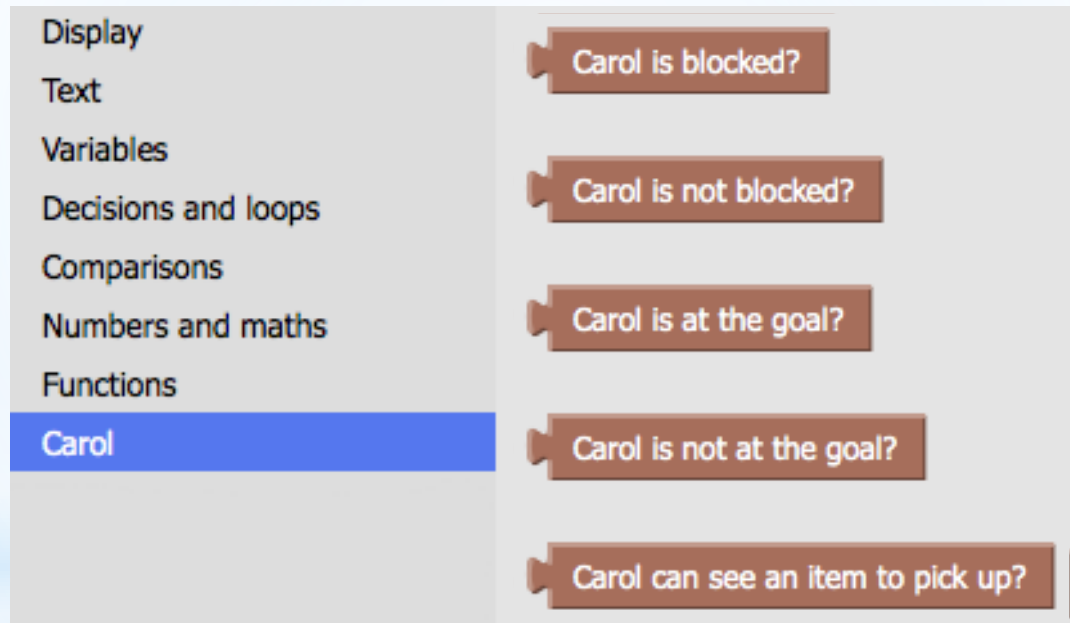


#1: Drag the condition which has to be true and clip it on here

#2: Drag the instructions which will happen if the condition is true into the gap here

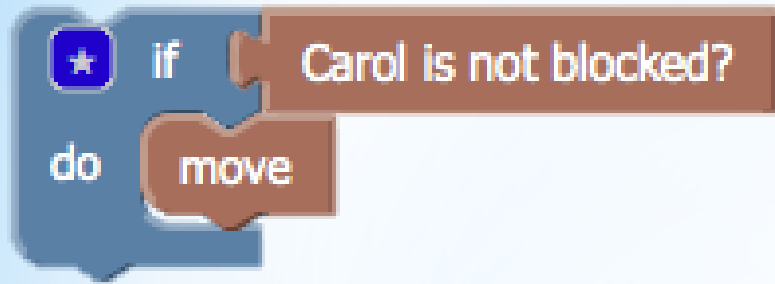
The *if* statement - conditions

- Carol has a number of things she can check for which you can use as a condition on your *if* statements

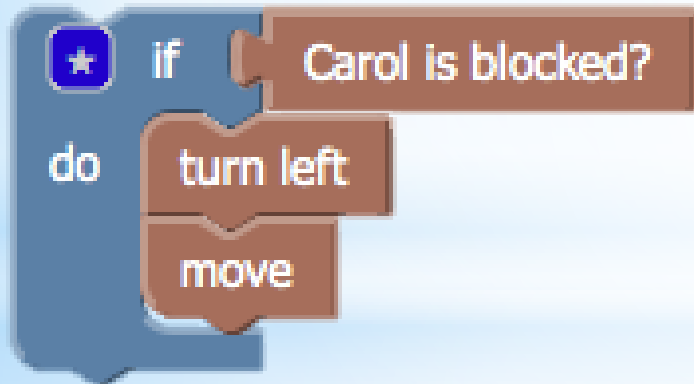


- Drag the condition you want and clip it onto your *if* block

The *if* statement - an example



- Only move if Carol isn't blocked (i.e. the path ahead is clear)

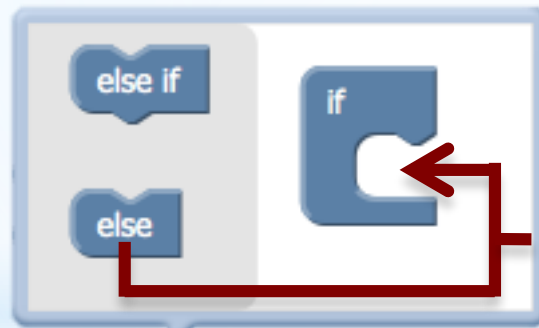
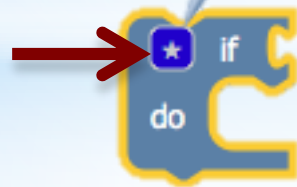


- If she's blocked, turn left before moving

if and else

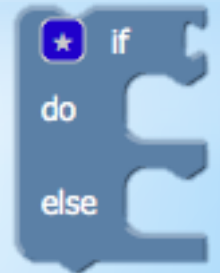
- ⦿ You can expand an *if* block using *else*
- ⦿ *else* allows you to give a second series of instructions that will happen if the condition is NOT true
- ⦿ To add an *else* to an *if* block:

#1: click the star -
the "speech bubble"
shown above will
appear



#2: drag the *else*
block in the speech
bubble into the *if*
block within the
speech bubble

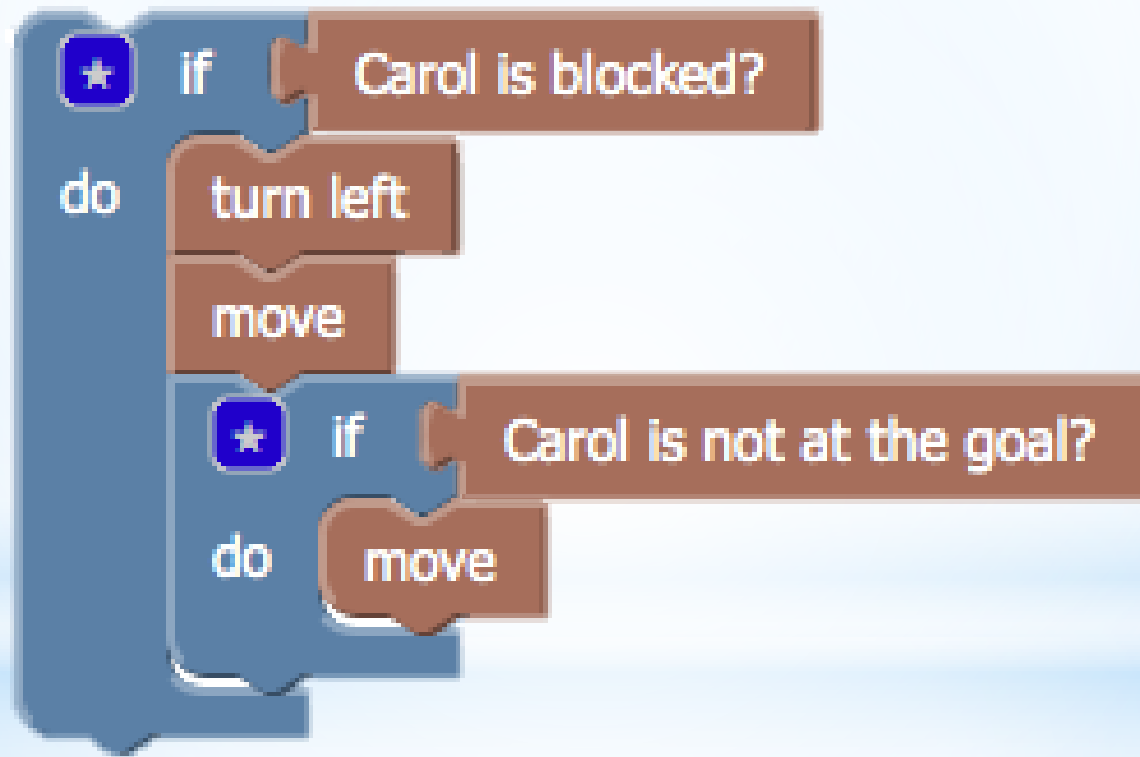
#3: the *if* block on
your canvas will
expand as shown
opposite



if and *else* - our previous example reworked

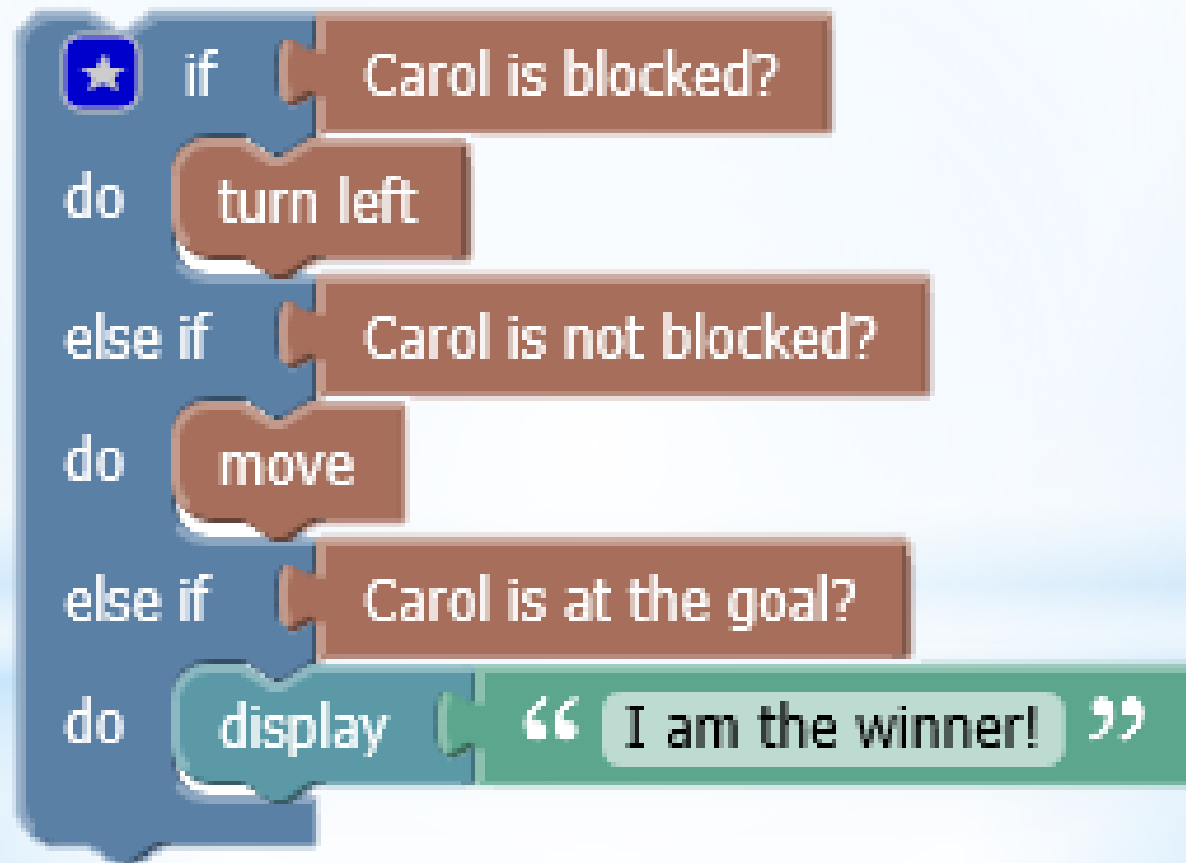


Remember that blocks can fit within blocks...

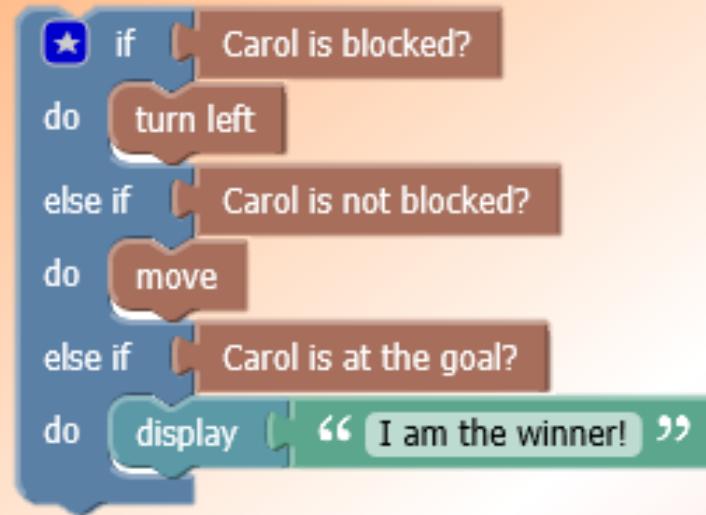
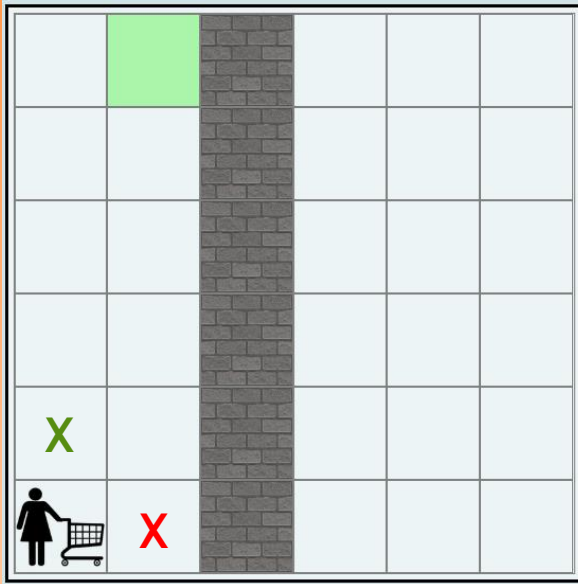


if, else and else if

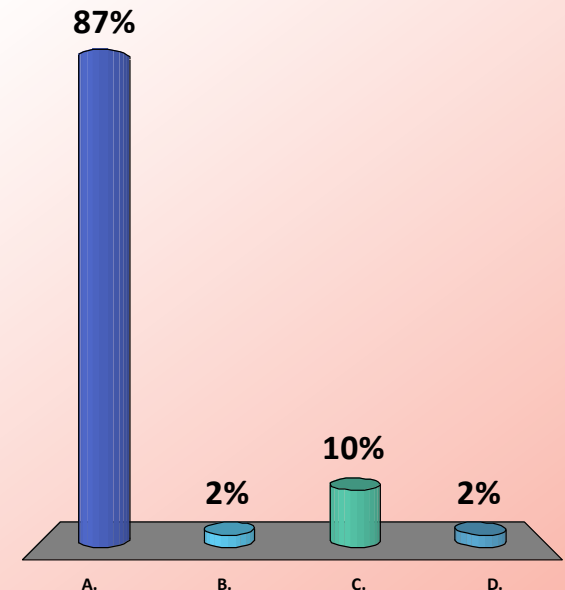
- ⦿ You can have a number of conditions by using *else if*:



Given the maze and block program shown below, what will Carol do?



- A. She will move into the square with the red X.
- B. She will not move but she will turn left
- C. She will turn left and then move into the square with the green X
- D. She will not move but the text **I am the winner!** will be displayed



Conditional loops

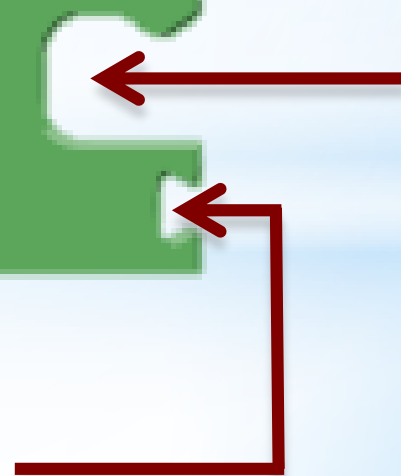
- ◉ Sometimes, you need to repeatedly do something until a condition is met
 - ◉ e.g. *keep moving until you encounter an obstruction*
- ◉ Programming languages all have loops which allow you to do this
- ◉ We have two blocks we can use in our Carol programs:



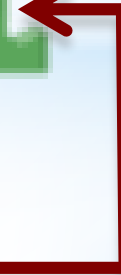
Drag the instructions which will repeat if the condition is true into the gap here



Drag the condition which has to be true for the instructions to repeat and clip it on here



Drag the instructions which will repeat if the condition is true into the gap here



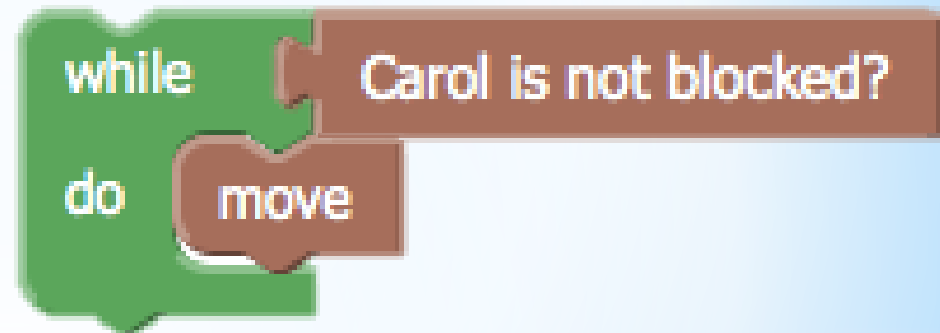
Conditional loops



- ⦿ There are two main differences between a *while* loop and a *repeat/until* loop
 - ⦿ #1
 - ⦿ A *while* loop checks the condition BEFORE the instructions are repeated
 - ⦿ A *repeat/until* loop checks the condition AFTER the instructions are repeated
 - ⦿ #2
 - ⦿ A *while* loop repeats as long as the condition is true
 - ⦿ A *repeat/until* loop repeats UNTIL the condition becomes true
 - ⦿ ...so it repeats as long as the condition is false

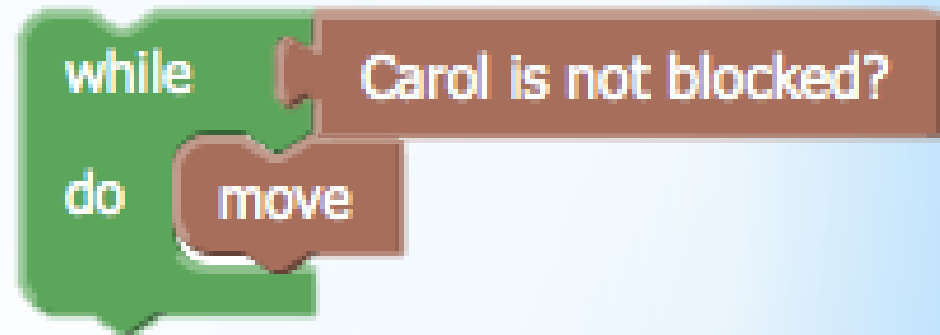
A practical example - "move until blocked"

- ⦿ Note how the condition is reversed... remember:
- ⦿ With *while* it repeats as long as the condition is TRUE
- ⦿ With *repeat/until* it repeats continuously UNTIL the condition becomes true - so it repeats as long as the condition is FALSE.

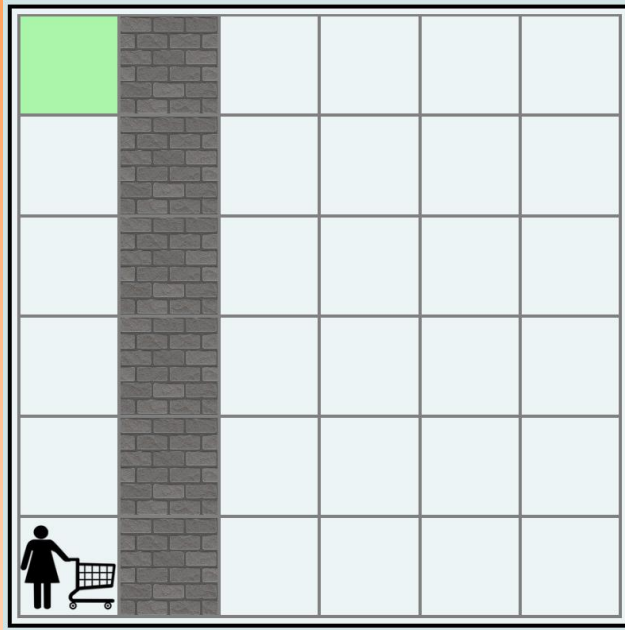


A practical example - "move until blocked"

- Other things to consider:
 - What is the minimum number of times each loop will repeat?
 - What happens first? Does it check the condition BEFORE or AFTER doing the command?
 - Also don't forget that you can put as many commands as you like inside the loop block! Our examples here have only one but we could have had loads and loads...



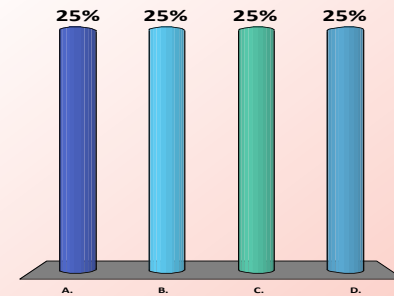
Given the maze below, what would be the effect of running the code blocks shown?



while Is Carol not blocked?
do move BLOCK #1

repeat move BLOCK #2
until Is Carol blocked?

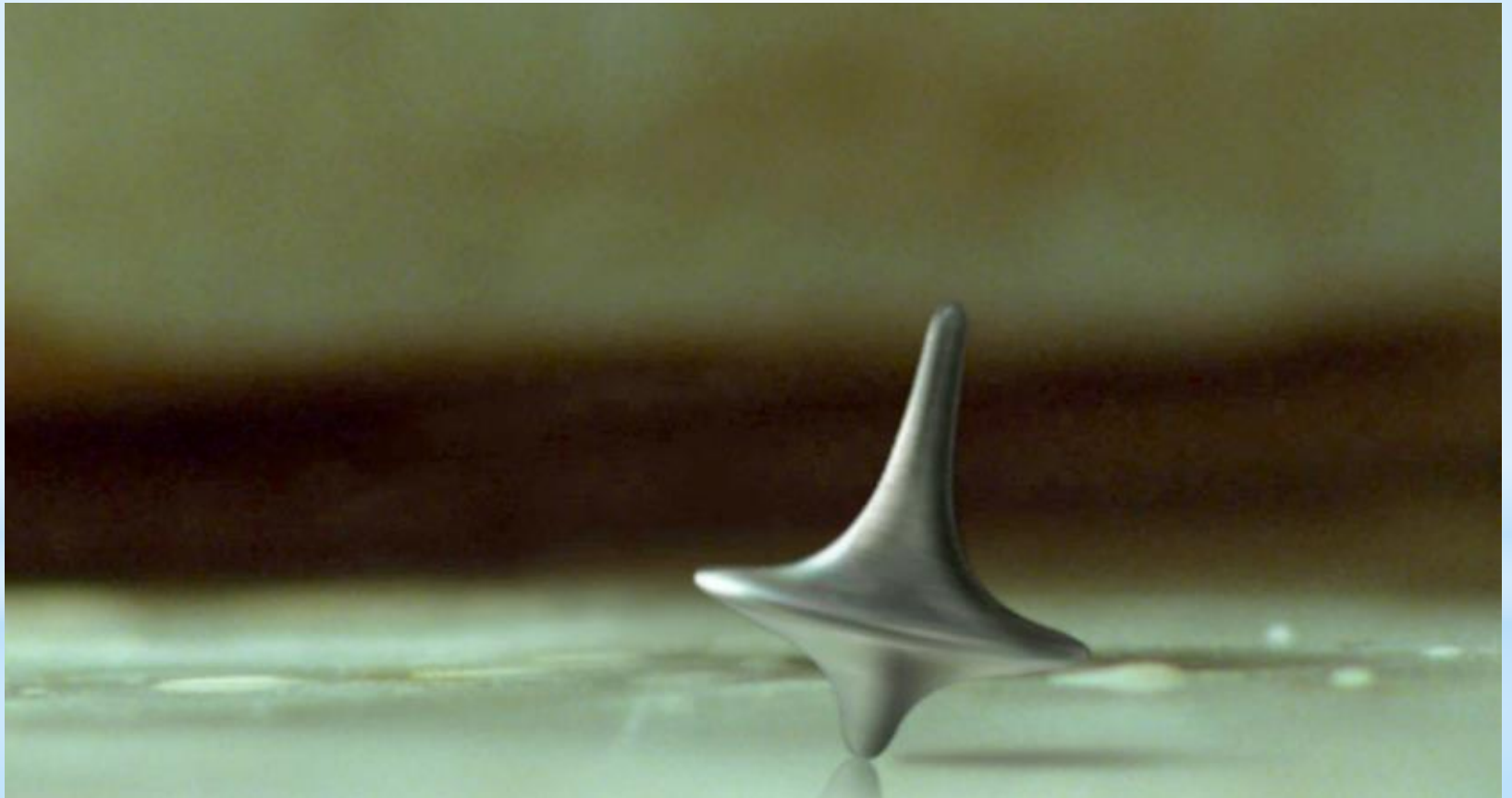
- A. Both blocks would run without an error
- B. Block #1 would run but block #2 would cause an error
- C. Block #2 would run but block #1 would cause an error
- D. Neither block would run without an error



Protip

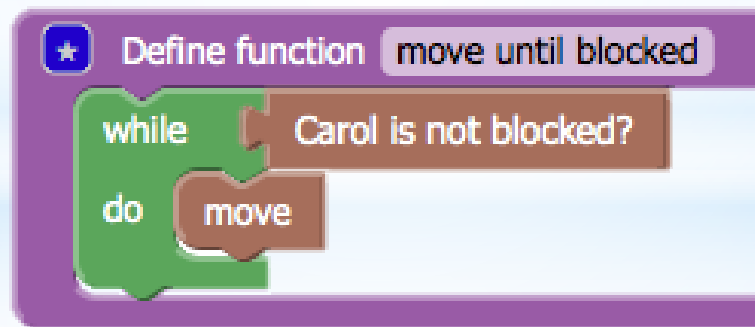
- ⦿ As you write code, you will inevitably find that there are certain functions that you use again and again across different solutions
 - ⦿ e.g. turn right, move until blocked, etc
- ⦿ Keep a "useful functions" file and as you create and find new functions, add it to the file
- ⦿ This will save a LOT of time in the future when you're building new solutions!

Recursion... something for the talented



Recursion... something for the talented

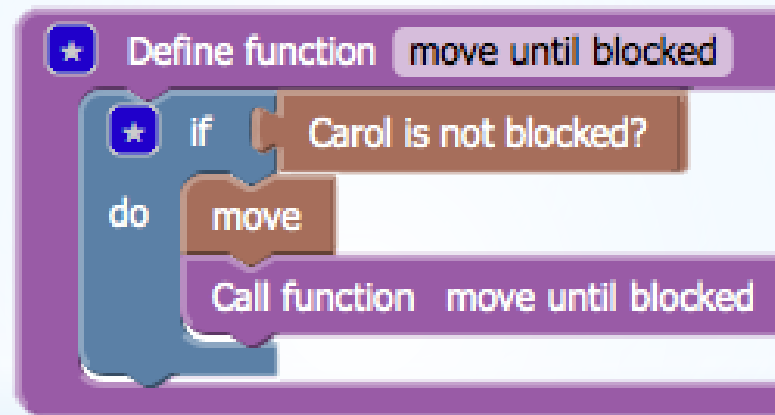
- ⦿ A recursive function is a function that calls itself
 - ⦿ (yes, this is a bit like the plot of the film *Inception*)
- ⦿ Consider if we wanted to write a function that would move Carol until she encountered a wall
- ⦿ We might use the while loop from the previous page in a function, e.g.



- ⦿ Recursion gives us another possibility...

Recursion... something for the talented

- Here is an example of the "move until blocked" function that uses recursion



- If you complete this weeks workshops and are at a loose end, try it out...
 - Can you figure out how it works?

Going beyond Carol...

The story so far

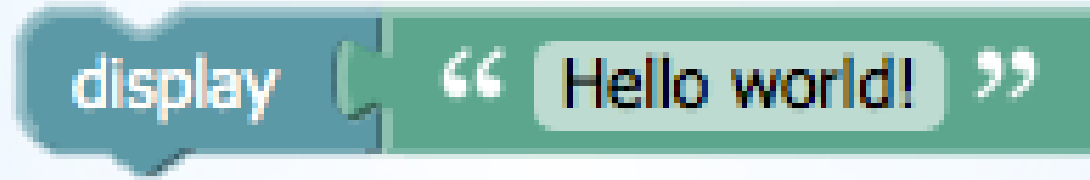
- ⦿ We have learned that there are a suite of basic programming constructs
 - ⦿ e.g. IF, FOR, WHILE, REPEAT, functions, etc
- ⦿ These constructs are like lego blocks
- ⦿ Some blocks fit together
- ⦿ Some blocks can clip inside other blocks
- ⦿ We clip lots of blocks together to make our programs

The story so far

- ⦿ Carol has up until now let us see the results of our programs in a visual way
- ⦿ We've seen the effects of repetition
 - ⦿ "Move ahead four squares"
 - ⦿ "Turn left three times"
- ⦿ We've been able to introduce some conditional processing
 - ⦿ "if the way ahead isn't blocked keep moving"
 - ⦿ "if the way ahead IS blocked turn left"
- ⦿ These are crucial programming concepts, but there are others you need to learn that don't really work well with Carol 😞
- ⦿ So... we are going to go up a level and start looking at more abstract problems

"Hello world!"

- ◉ Virtually every beginner's course in programming or every tutorial in a given programming language starts out with a simple program to print *Hello world!* on the screen
- ◉ We won't break with tradition, so...

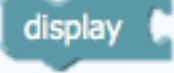






raw code version:

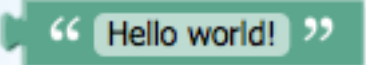

```
display "Hello world!"
```

"Hello world!"


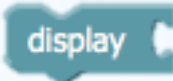
⦿ Composing the blocks for *Hello World*:

1. Choose the  block from the **Display** menu (surprisingly enough!) and drag it onto your canvas
2. Choose the  block from the **Text** menu and drag it onto your canvas
3. Drag the  block and clip it onto the side of the  block
4. Click between the quotes of the  block. You will get a flashing cursor and will be able to type text. Type *Hello world* into this area

About the blocks of "Hello World"

- ◉ The  block is what is called a **constant**
- ◉ This means it does not change (it is constant!)
- ◉ In this example, this constant is text based
 - ◉ The text between the quotes is the *value* of the constant
- ◉ If we clip a text constant to a  block, the result will be to print that text to the screen
- ◉ In programming, you will often hear the term *string* used to describe text
 - ◉ (this is derived from the phrase "a string of text")
- ◉ Get used to hearing the word *string* and using it yourself - it is part of the basic terminology of programming

Number constants

- ◉ We can also have constants that are numeric
- ◉ A numeric constant can be found on the **Numbers and Maths** menu in the form of the  block
- ◉ Click on the **0** to change the number
- ◉ You can clip a numeric constant on the  block too, e.g.



What is the difference?



- ⦿ Is there a difference between the two block combinations?
 - ⦿ look at the colour coding
 - ⦿ look at the quotes...
- ⦿ Will there be any difference in what gets output?
- ⦿ Is this difference important?

Introducing variables

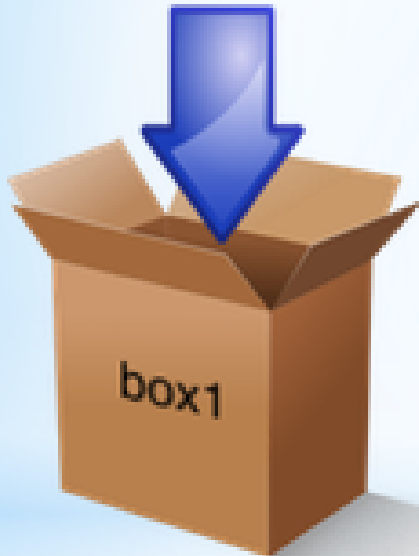
- ⦿ Constants are all well and good, but we need computers and the programs that run on them to be able to handle things that change... e.g.
 - ⦿ The current score in a computer game
 - ⦿ The current speed of your car
 - ⦿ The current temperature in your heating system
- ⦿ The clue is in the name; a *variable* is a piece of information that varies, i.e. it can change

Introducing variables

- ⦿ A variable in programming is like a box for data
- ⦿ You can put things in the box
- ⦿ Each box is labeled
 - ⦿ This means that you can tell which box is which
 - ⦿ This means that you can refer to a specific box when you need to

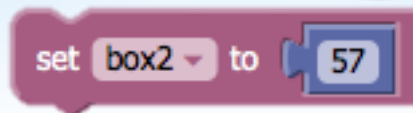
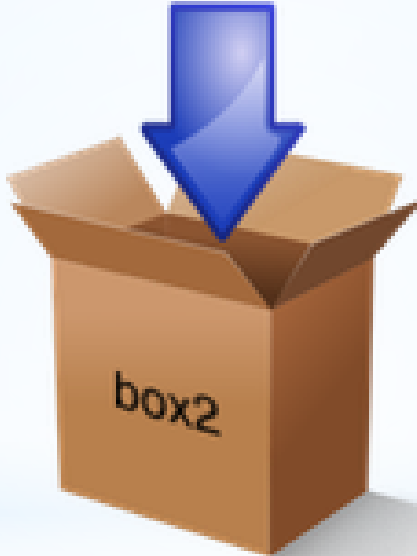
Assigning a value to a variable

"Paul"



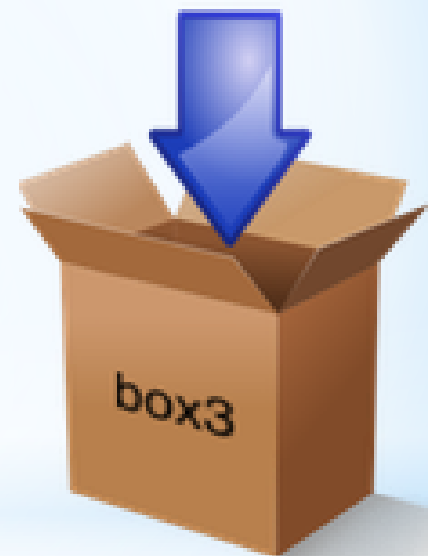
raw code version:
`set box1 = "Paul"`

57



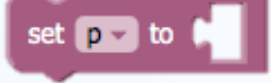

raw code version:
`set box2 = 57`

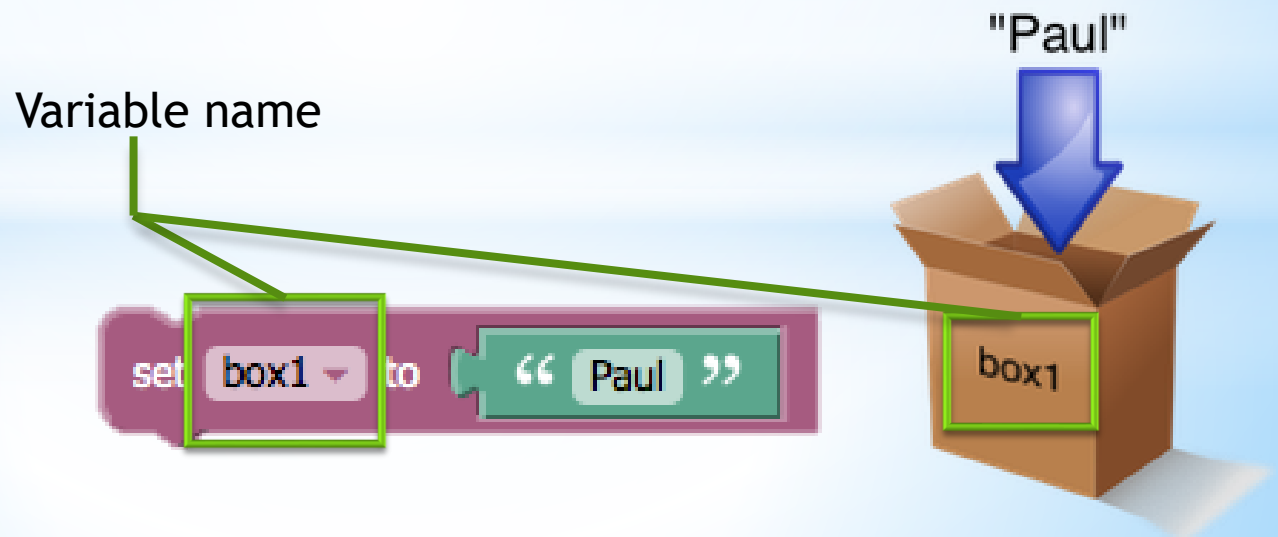
-2.7521



raw code version:
`set box3 = -2.7521`

Using the variable blocks

- From the **Variables** menu, drag the  block onto the canvas
- It will use a variable name of **p** as a default...
 - ...this is a bit rubbish...
 - ...you should change the variable name by clicking on the  and then selecting **New Variable**
- The variable name is like the label on the side of the box:



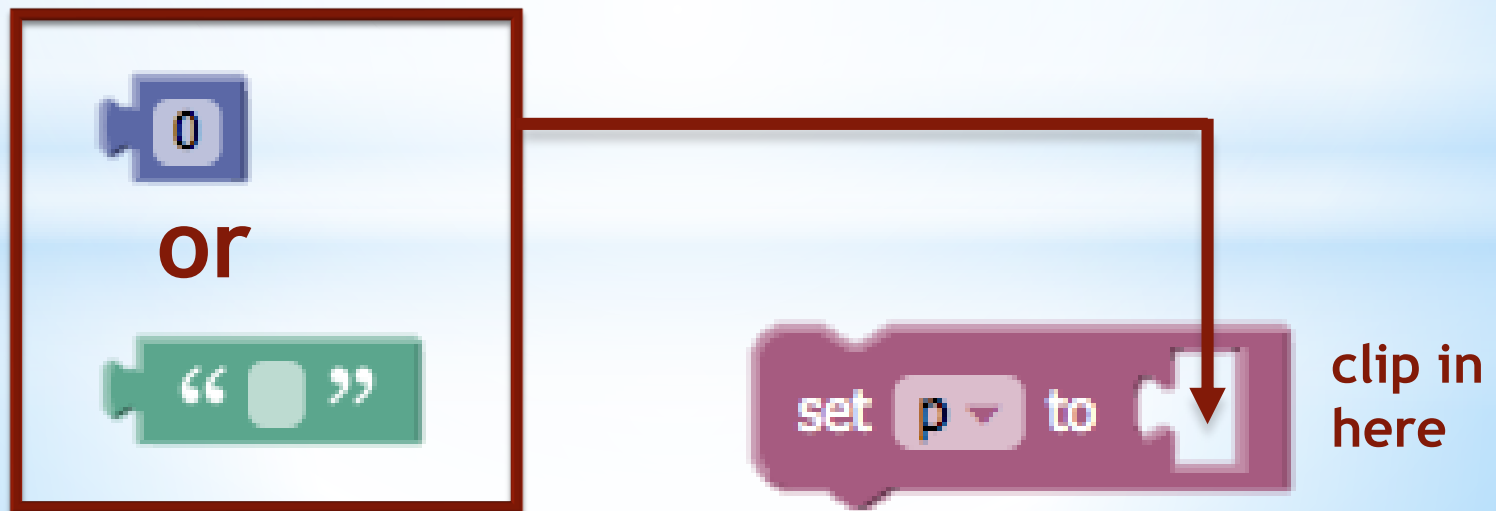
Using the variable blocks

- ⦿ About variable names:
 - ⦿ You should always choose a variable name that tells you what the variable is being used for
 - ⦿ Most programming languages have restrictions on the letter and symbols (characters) you can use in variable names
 - ⦿ If you avoid using spaces, avoid having numbers at the start of the variable name, and avoid anything that isn't a letter or a number you won't go far wrong
- ⦿ Good variable names:
 - ⦿ counter, score, position, cost, costOfItem, currentTemperature
- ⦿ Bad variable names
 - ⦿ p, myVariable, var1, box1

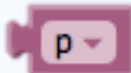

NB: Yes, "box1" would be a bad variable name! Yes, I've been using it on these slides - but only to reinforce the concept of variables being like boxes. The name "box1" tells us nothing about what the variable is *for* or what we're storing in it - therefore it would be a rubbish name to use in real programs!

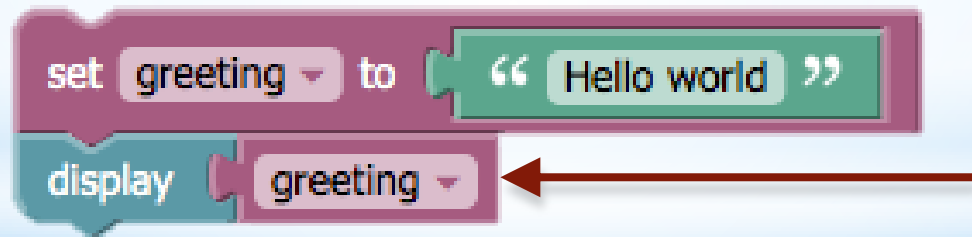
Using the variable blocks

- ⦿ You also need to specify what you want to put into the box
- ⦿ This can be a constant, another variable, or a maths expression (which we'll see later)
- ⦿ For now, let's drag a constant (either a text or number one) into the empty space of your variable block
- ⦿ Don't forget to give your constant a value - don't leave it as zero or as an empty text string (unless that's what you want)



Using the variable blocks

- ⦿ You can retrieve what's in your box and make use of it too
- ⦿ In the **Variables** menu, find the  block and drag it onto your canvas
- ⦿ Click the  and select the name of the variable you want to refer to
- ⦿ You can clip a variable block onto any other blocks where it fits, for example:

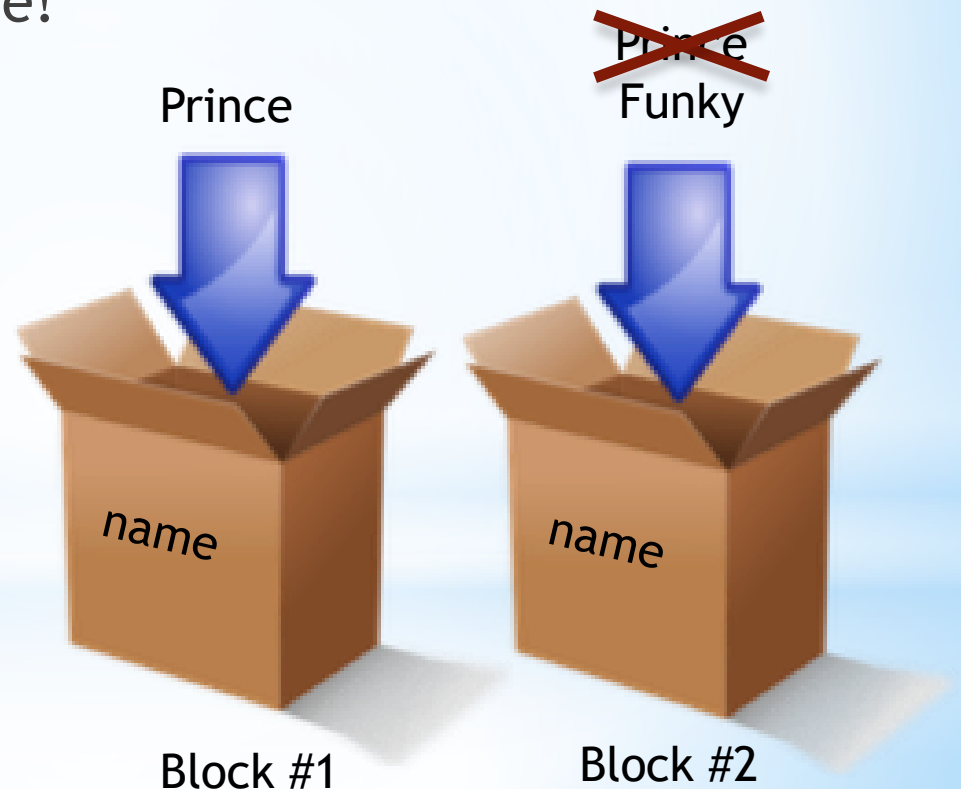


Variable block is
clipped onto the
display block

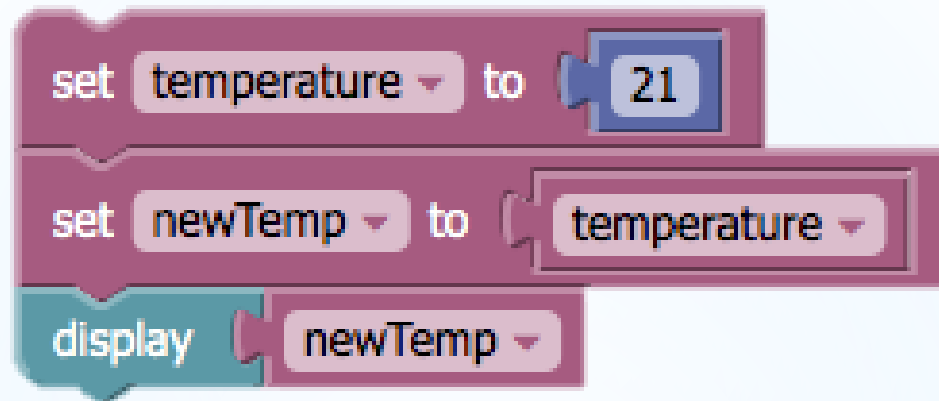
- ⦿ What would the result of running this code be?

Variables contain only ONE value at a time

- ⦿ If you assign a new value to a variable, any existing value is replaced with the new one!



If the block fits, use it...



- ⦿ What will be in the variable *newTemp*?
- ⦿ What will be displayed on the screen?

The FOR loop - a reprise

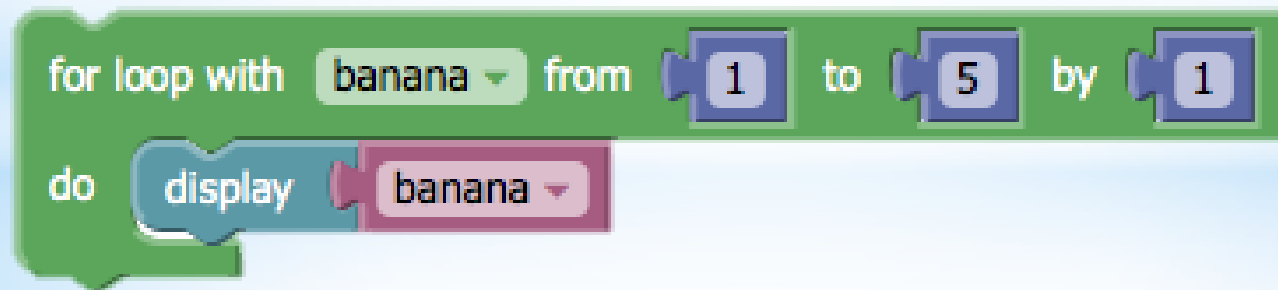
- Remember how we used the FOR loop as a way to repeat Carol moves a set number of times, e.g.



- The FOR loop also involves defining and making changes to the contents of a variable

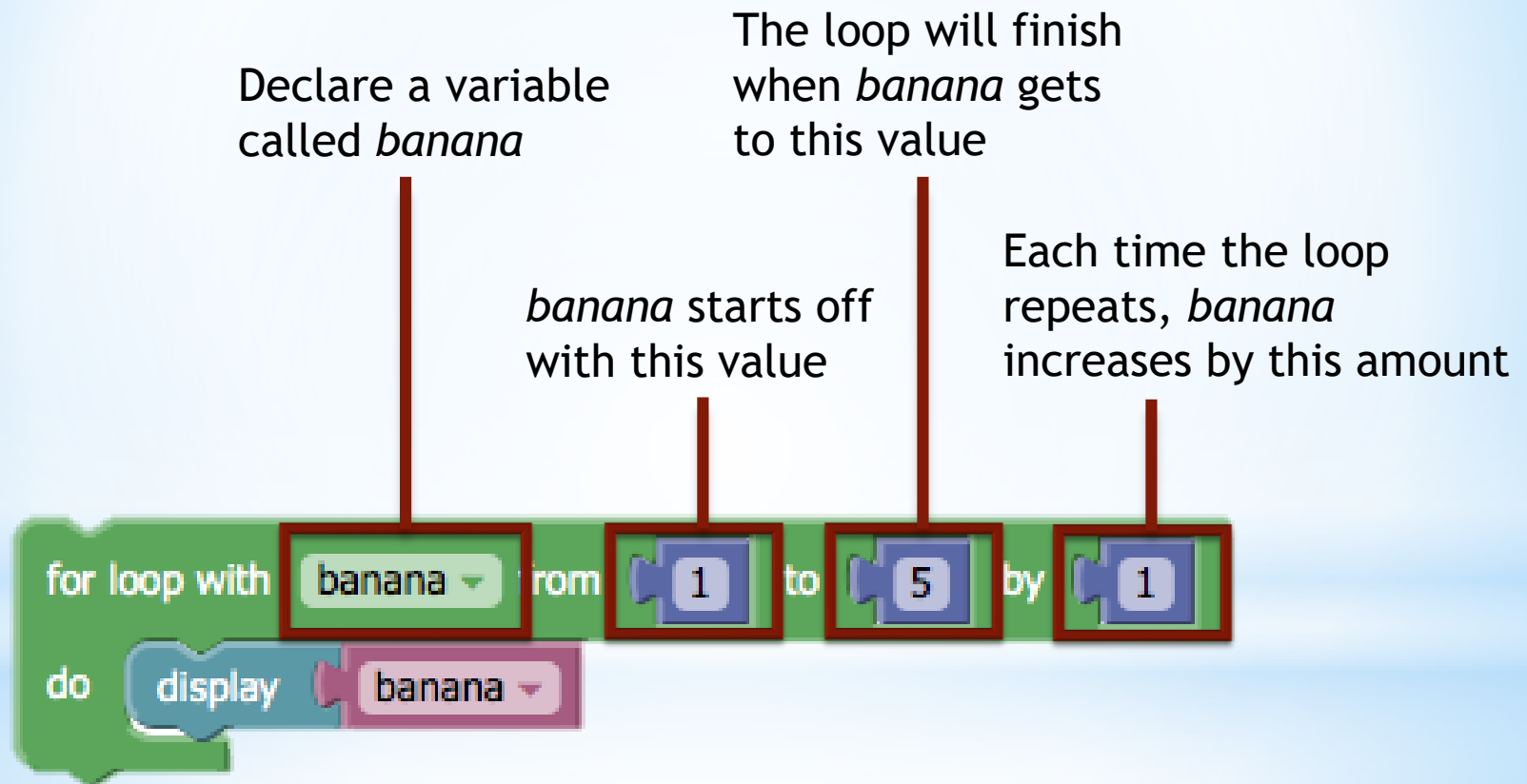
The FOR loop - a reprise

- Here's a non-Carol example:



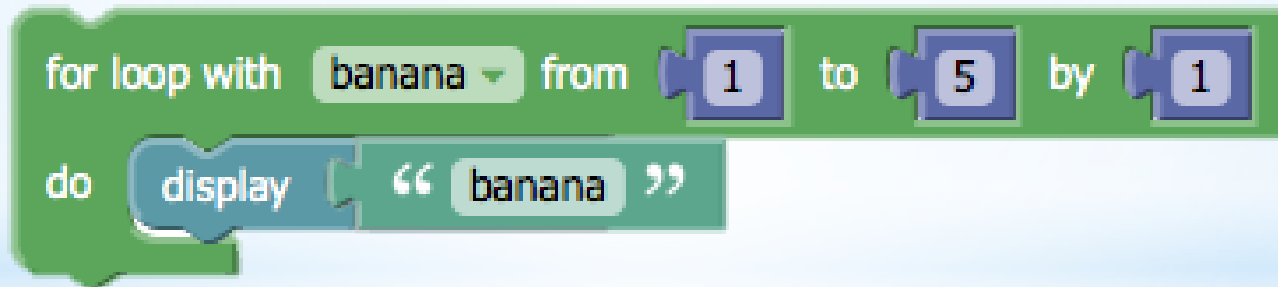
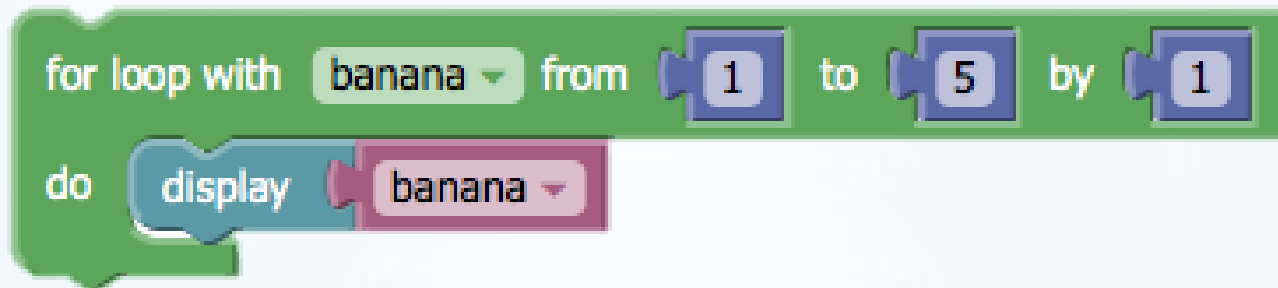
The FOR loop - a reprise

- Here's a non-Carol example:




The FOR loop - a reprise

- ⦿ "If the block fits..."



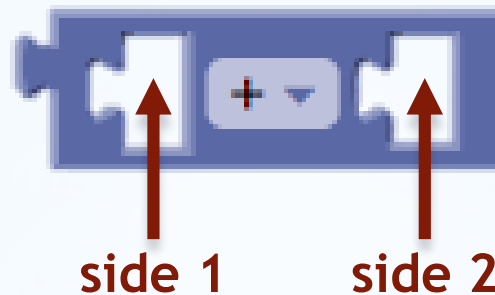
- ⦿ What's the difference? Why?
- ⦿ PS: don't forget to name the variable in your loop when you drag a new FOR loop block into your programs...



Performing calculations

- ⦿ Most programs will involve some form of mathematical calculation
- ⦿ Most programming languages use the same set of symbols to perform the basic mathematical operations
 - + for Add - for Subtract
 - * for Multiply / for Divide
 - ^ for Powers
- ⦿ To perform a calculation, we use the  block from the **Numbers and Maths** menu

Using the calculation block

- There are two sides to the calculation block



- The "holes" in the block on each side have to be filled
- You can fill them with
 - numeric constants 
 - variables (that contain numbers) 
 - other calculations
- You should also select which operator you need (i.e. + - / * ^)

Using the calculation block

- When you have composed a calculation, you need to do something with the result

- You might

- print the result to the screen



- put the result into a variable



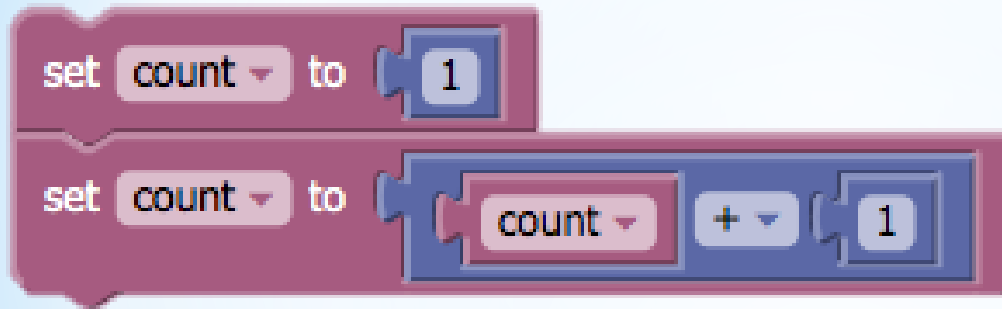
- use the result in another calculation by nesting a calculation block within another calculation block



- Use the result as part of the condition for a FOR loop!



Modifying variable values with the calculation block



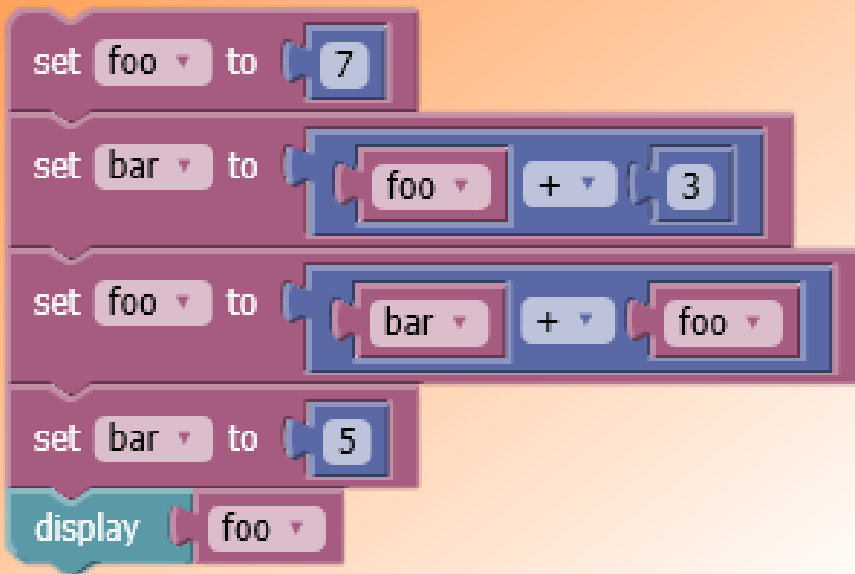
raw code version:

```
set count = 1
```

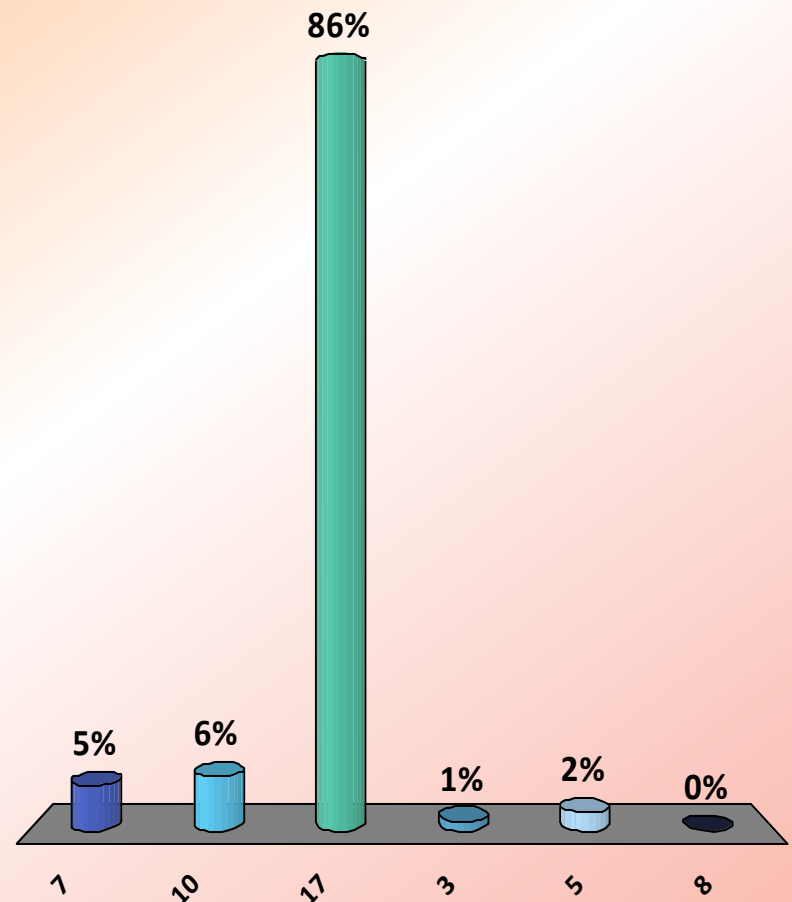
```
set count = count + 1
```

- ⦿ The first block sets the **count** variable to 1
- ⦿ The second block increases the value of the **count** variable by 1
 - ⦿ Why? How?
- ⦿ The moral of the story - you can refer to the variable you're setting when you calculate a new value for it

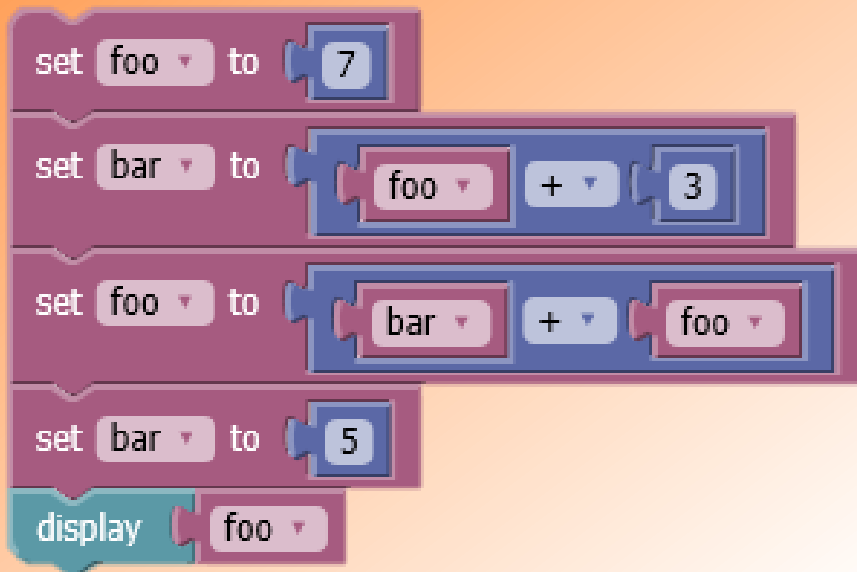
What will be printed if the following blocks were run?



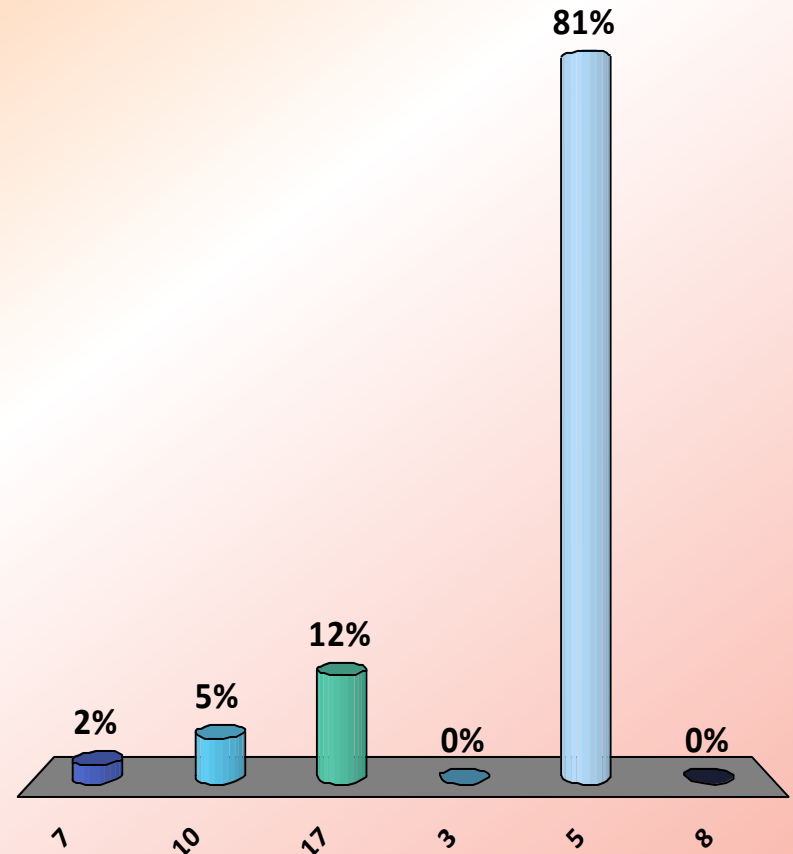
- A. 7
- B. 10
- C. 17
- D. 3
- E. 5
- F. 8



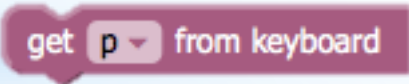
At the end of the program, what will be stored in the variable named bar?



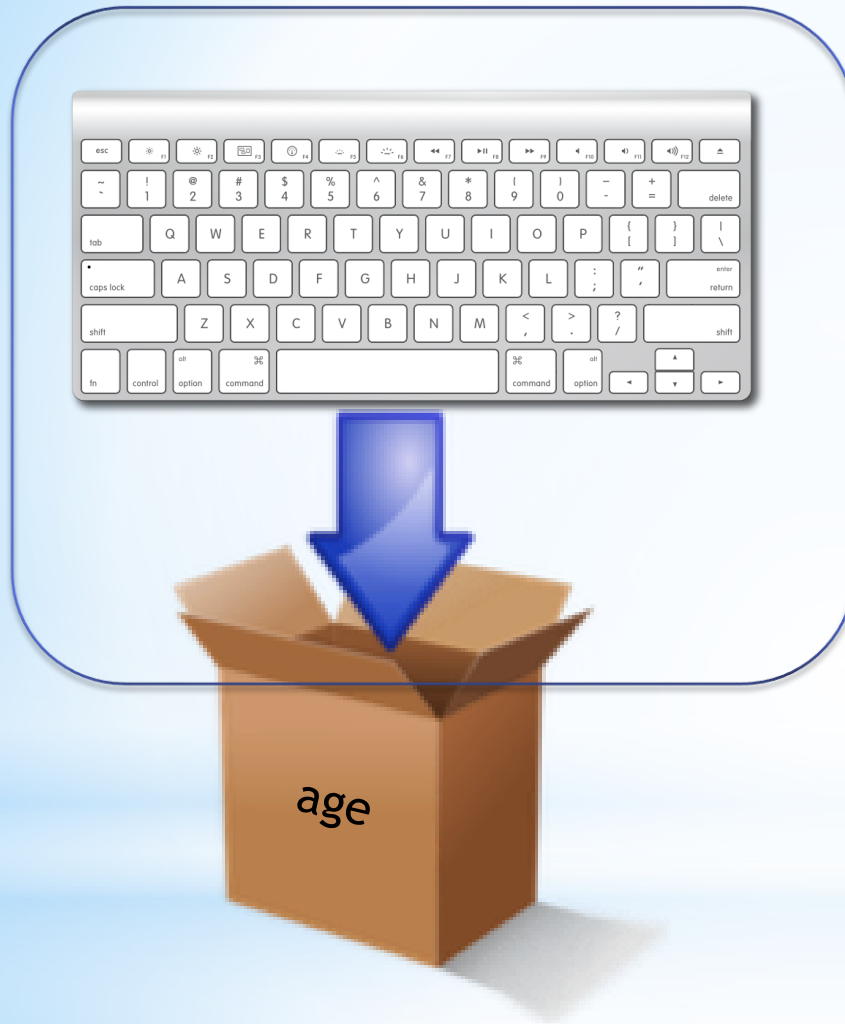
- | | |
|-------|------|
| A. 7 | D. 3 |
| B. 10 | E. 5 |
| C. 17 | F. 8 |



One last thing... getting input from the user

- ⦿ Computer programs (usually) need to interact with the user in some way
- ⦿ Computers (usually) have some kind of device attached that lets the user interact with the machine
 - ⦿ e.g. a keyboard
- ⦿ In your workshop exercises, you will write programs that get the user to input data from the keyboard
- ⦿ Use the  block to do this
 - ⦿ This will pause the program and wait for the user to type something followed by the return key
 - ⦿ The value they type on the keyboard will then go into the variable
 - ⦿ Don't forget to name your variable something better than **p**!)

In a nutshell



get **age** from keyboard

Summary

- ⦿ *Functions* allow us to define a set of instructions and then *call* them later when we need to do this set of instructions
- ⦿ A function has a name
- ⦿ When we call the function, we refer to this time
- ⦿ Functions **ONLY** run when they're called
 - ⦿ ...even if they're declared at the start of the program, they don't run until (and unless) they're called
- ⦿ You can think of functions as a way of building new commands using the existing ones
- ⦿ The *main body* of the program is the part of the program not enclosed by functions - the first line of this is where the program will start
- ⦿ When you have a function that calls itself, this is called *recursion*

Summary

- ⦿ As well as functions, there are several other fundamental constructs in programming that allow our programs to make decisions and to repeat instructions
 - ⦿ The *if* statement makes a one-off decision
 - ⦿ The *for* loop repeats a set of instructions a specific number of times
 - ⦿ The *while* loop and the *repeat/until* loop repeat a set of instructions until a specified condition is met
- ⦿ Virtually all programming languages will have some form of these fundamental constructs...

Summary

- ⦿ A variable is like a box
 - ⦿ The box has a label (the variable name)
 - ⦿ You can store one thing in the box at a time
- ⦿ You can modify what's in the box with calculations