LAST WEEK

- We got to know each other a little bit
- We loaded up tinkercad
- A basic framework for how to design code
- We learnt how to make a really simple circuit

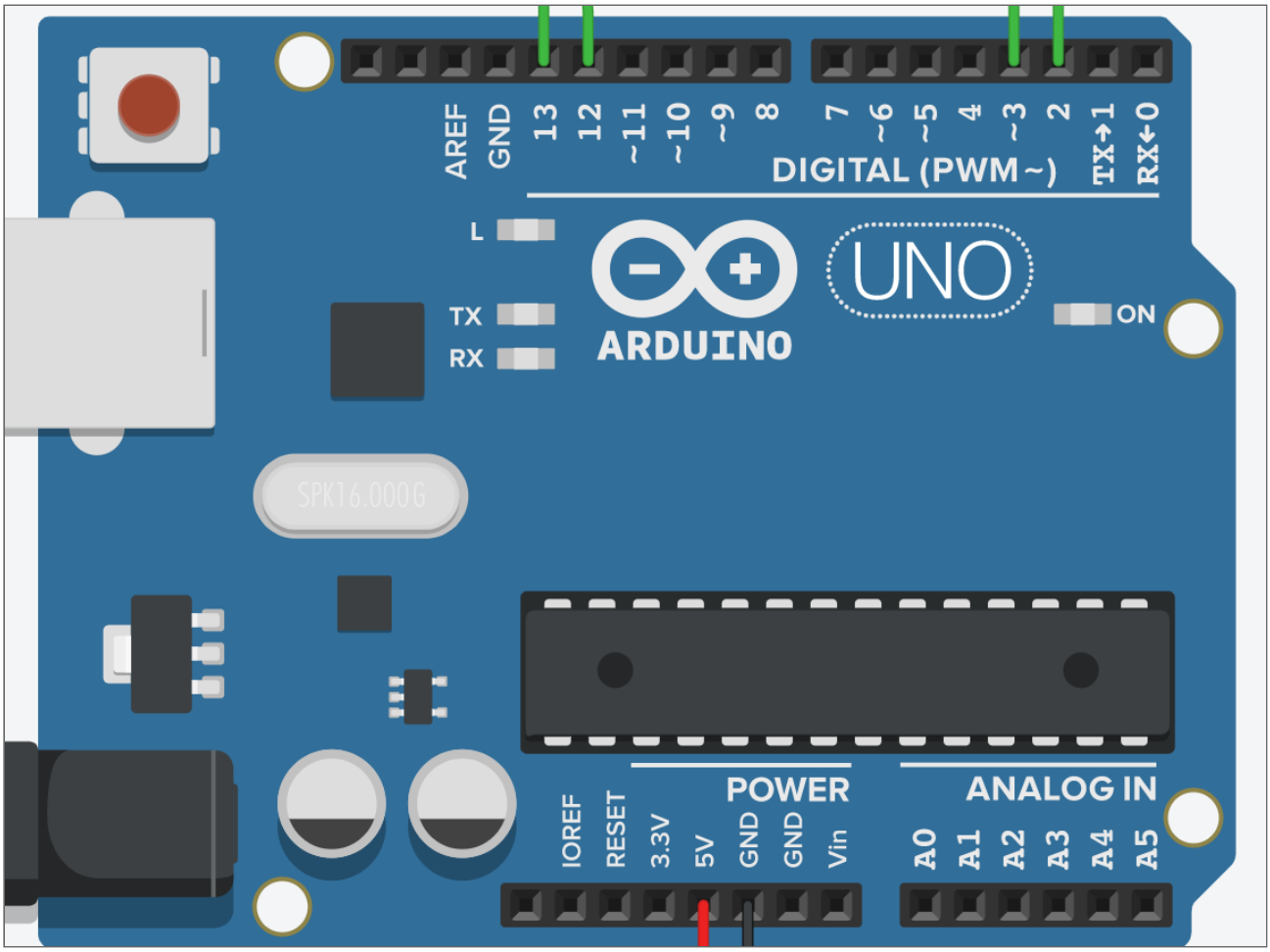
THIS WEEK

- Rules of a circuit
- How a LED works
- How buttons work

HOW DO CIRCUITS WORK ON ARDUINOS?

- An Arduino circuit uses electricity to perform a task
- The Arduino board acts as the brain of the circuit, controlling the flow of electricity
- Different components, such as lights, motors, and sensors, can be connected to the board
- A program is written using the Arduino software to tell the board what tasks to perform and how to control the flow of electricity
- The program is transferred from the computer to the Arduino board
- The program runs on the board and controls the circuit.

THE ARDUINO


AREF
GND
13
12
~11
~10
~9
8
7
~6
~5
4
~3
2
TX→1
RX←0
DIGITAL (PWM~)
L
TX
RX
ARDUINO
UNO
ON
SPK16.000G
POWER
ANALOG IN
IOREF
RESET
3.3V
5V
GND
GND
Vin
A0
A1
A2
A3
A4
A5

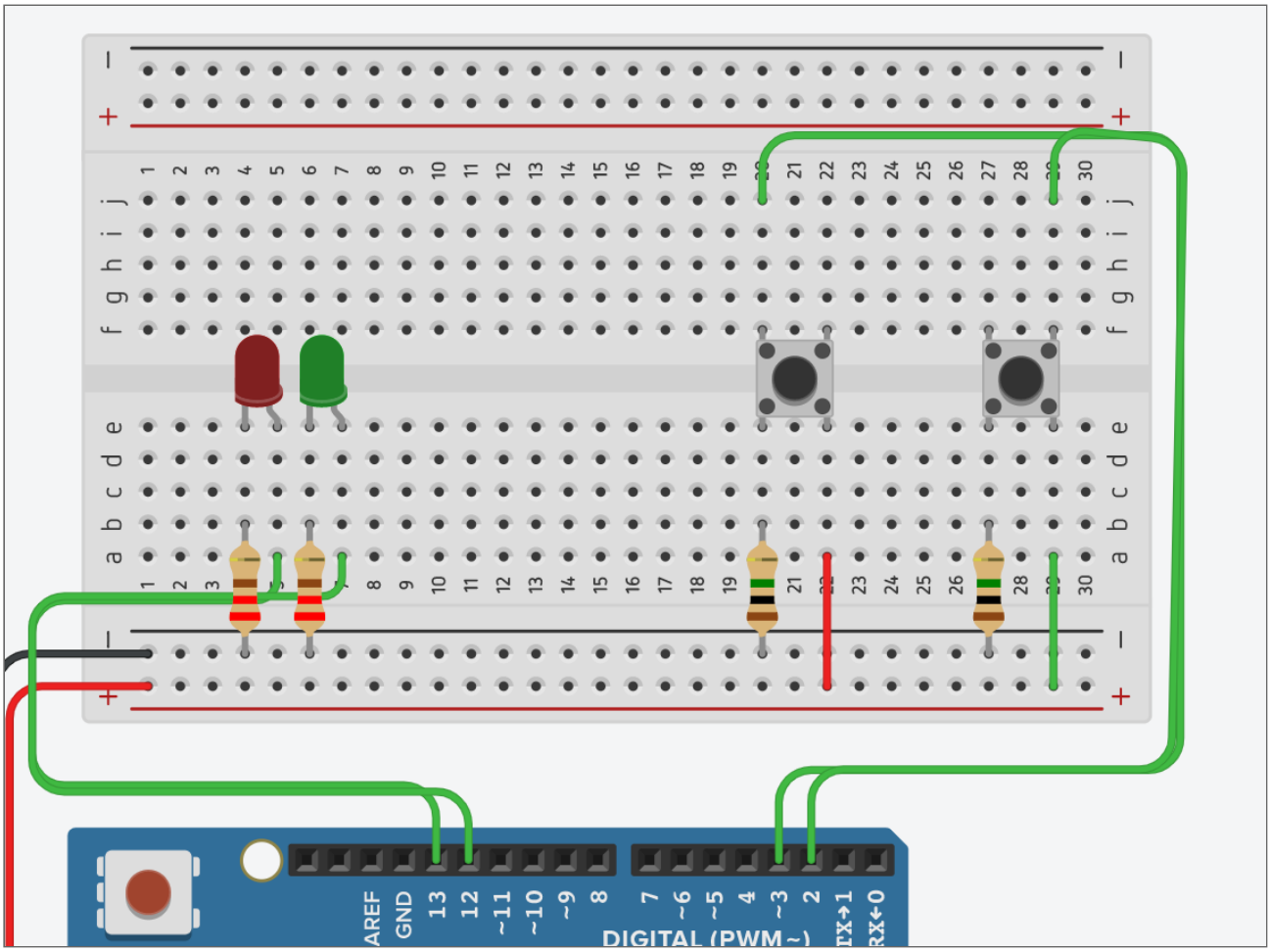
RULES OF A CIRCUIT

- Power flows a bit like fluid through a tube (kinda but not really)
- Power effectively flows from + to - (actually, the reverse)
- All components must share a common ground
- Many components need to limit the flow of current with resitors

HOW DO BREADBOARDS WORK?

- Holes: A breadboard has a grid of holes into which electronic components can be inserted and connected.
- Conductive strips: The holes are connected internally by metal strips, allowing for a electrical connection to be made between components that are inserted into adjacent holes.
- Temporary connections: The connections made on a breadboard are temporary, making it easy to test and change a circuit without having to solder components together.
- Power buses: Most breadboards have power buses running along the sides, providing a convenient way to distribute power to multiple components in the circuit.
- No soldering required: No soldering is required when using a breadboard, making it a good choice for beginners and those who want to avoid the hassle of soldering.

BREADBOARDS



READING RESISTORS

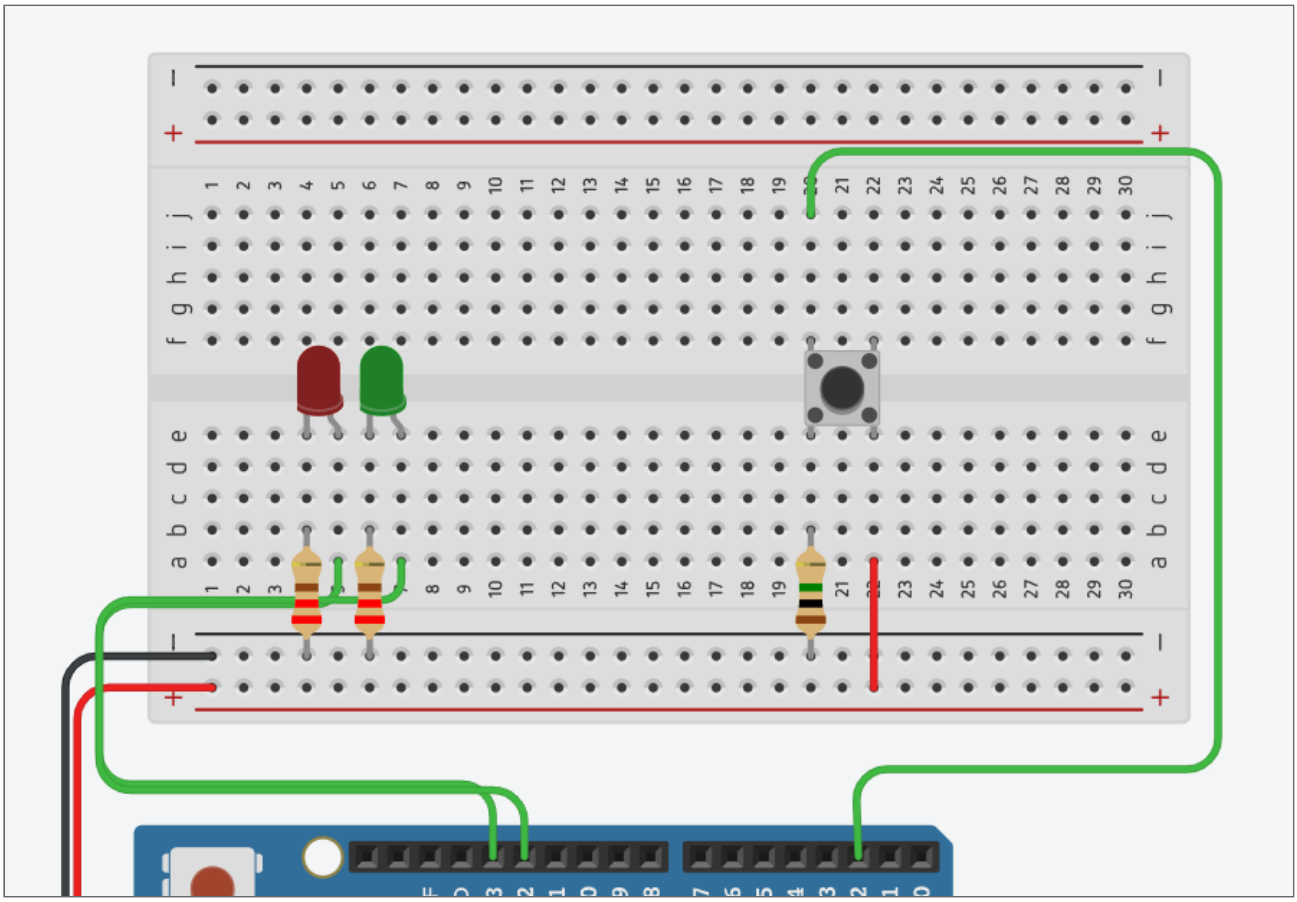
- Color-coding: Most resistors in Australia use color-coding to indicate their resistance value.
- Bands: The color-coding is represented by bands of color on the resistor, typically 4 or 5 bands.
- Band order: The first two or three bands represent the first two or three digits of the resistor value, while the last band represents the number of zeros to be added to the end of the value.
- Band order: The first two or three bands represent the first two or three digits of the resistor value, while the last band represents the number of zeros to be added to the end of the value.
- Color chart: A color chart can be used to decode the resistance value from the resistor's color bands.
- Tolerance: Some resistors may also have a tolerance band, which represents the range of acceptable resistance values for the resistor.
- Measurement units: The resistance value is usually measured in ohms (Ω).
- You read them from left to right

BUT WE DON'T HAVE TO DO THE MATHS OURSELVES:

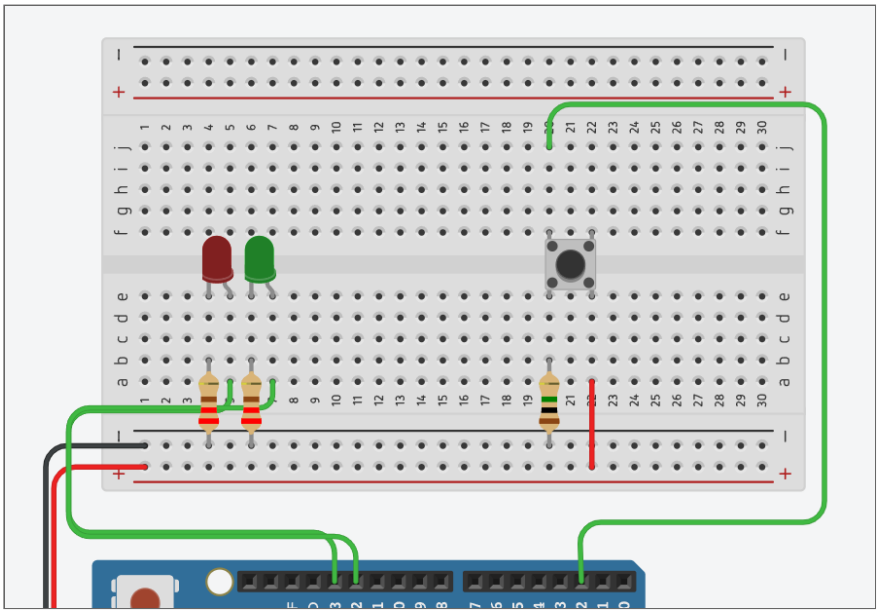
https://www.digikey.com.au/en/resources/conversion-calculators/conversion-calculator-resistor-color-code

BUTTONS

Consider the following circuit

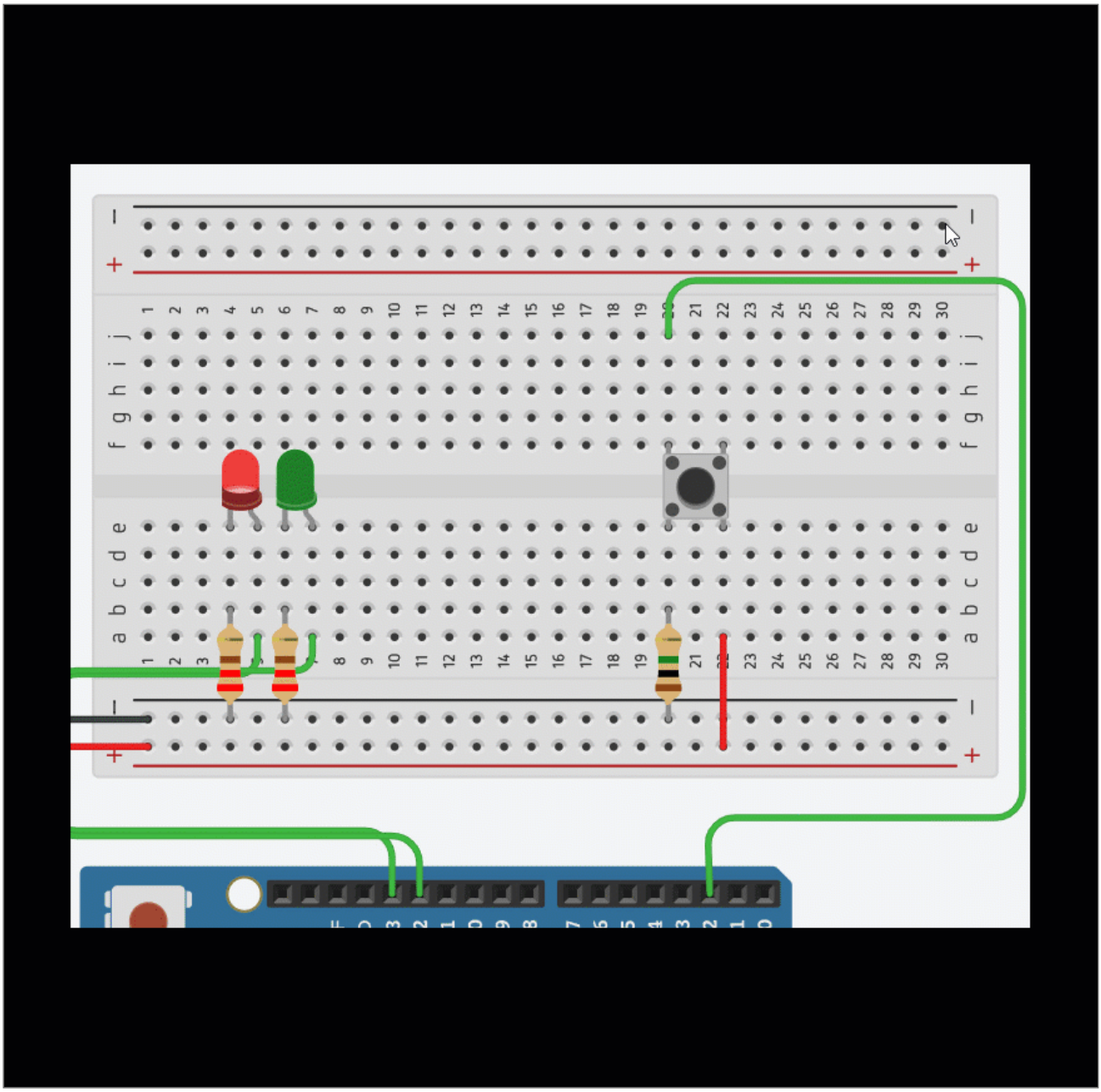


DEFINE THE PROBLEM

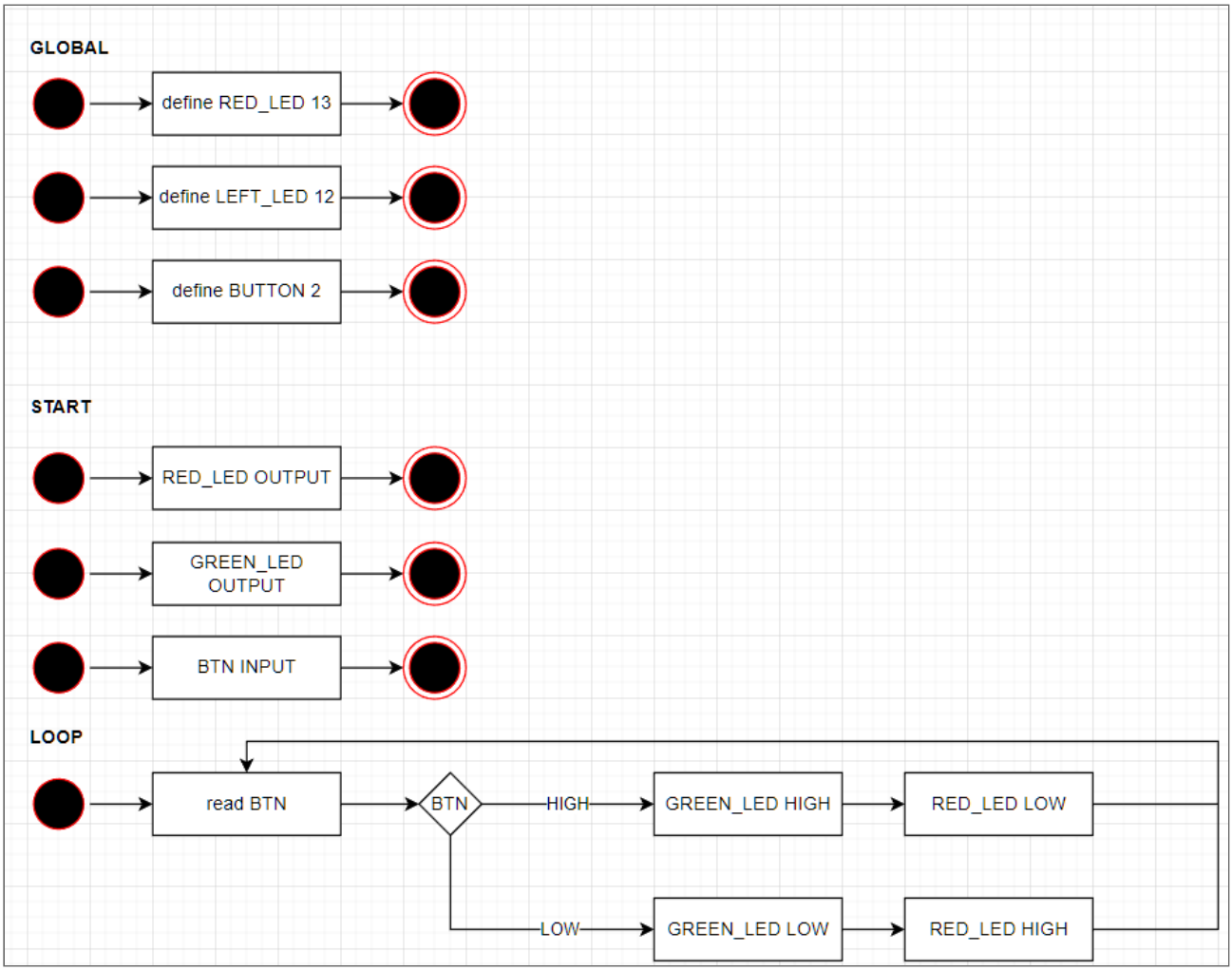


- By default red is on
- When you press the button turn red off and green on
- when button is released turn green off and red on

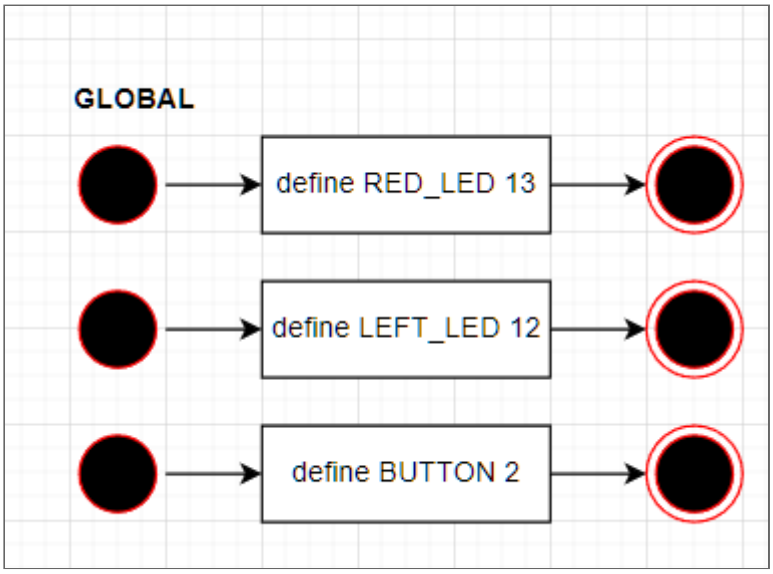
HERE'S A PICTURE!



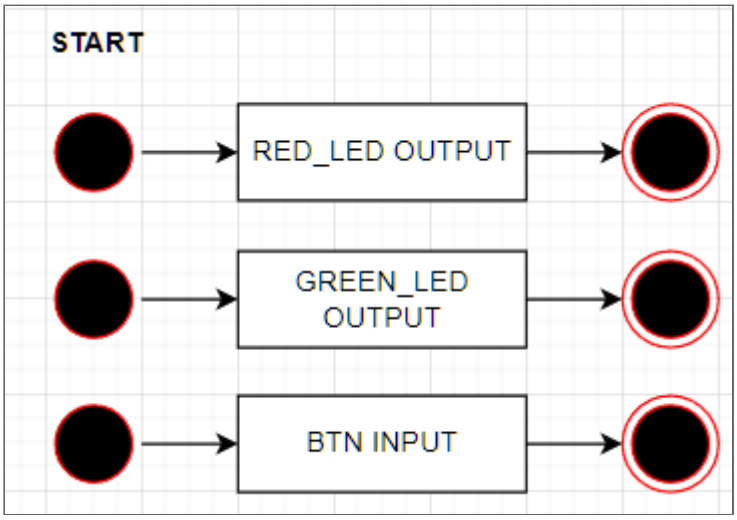
BUTTON DESIGN



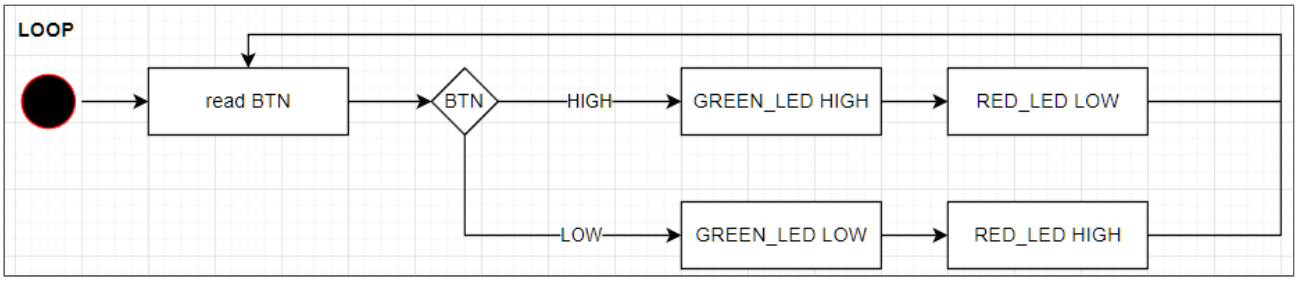
GLOBAL



START



LOOP

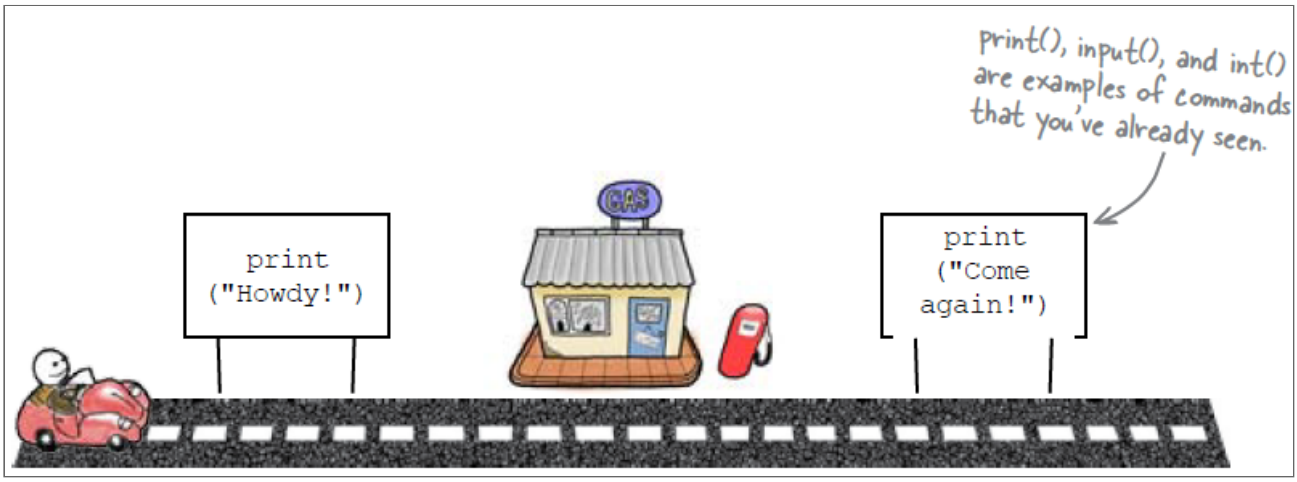

LOOP
read BTN
BTN
HIGH
GREEN_LED HIGH
RED_LED LOW
LOW
GREEN_LED LOW
RED_LED HIGH

CODING CONDITIONS

```
#define LED_RED 13
#define LED_GREEN 12
#define BUTTON 2

int btn;

void setup()
{
    pinMode(LED_RED, OUTPUT);
    pinMode(LED_GREEN, OUTPUT);
    pinMode(BUTTON, INPUT);
}

void loop()
{
    btn = digitalRead(BUTTON);
    if (btn) {
        digitalWrite(LED_RED, LOW);
        digitalWrite(LED_GREEN, HIGH);
    } else {
        digitalWrite(LED_RED, HIGH);
        digitalWrite(LED_GREEN, LOW);
    }
}
```

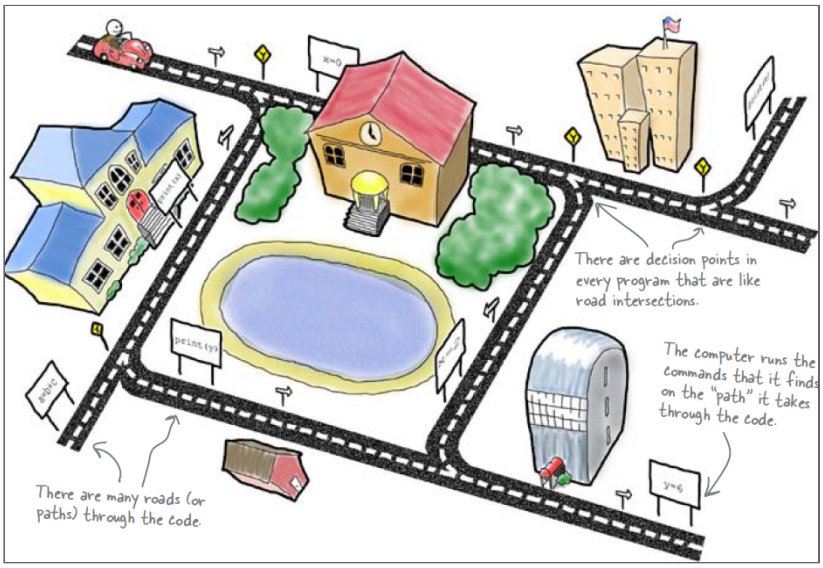
A PROGRAM IS MORE THAN JUST A LIST OF COMMANDS

You could create a program that was simply a list of commands. But you almost never will. This is because a simple list of commands can only be run in one direction. It's just like driving down a straight piece of road: there's really only one way of doing it.

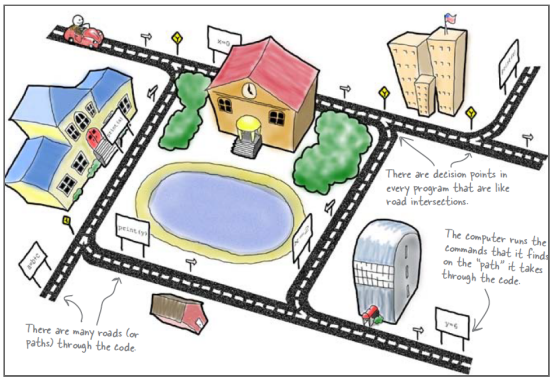


YOUR PROGRAM IS LIKE A NETWORK OF ROADS

Programs need to do different things under different circumstances. In the game, the code displays “You win!” if the user guesses the number correctly, and “You lose!” if not. This means that all programs, even really simple programs, typically have multiple paths through them.



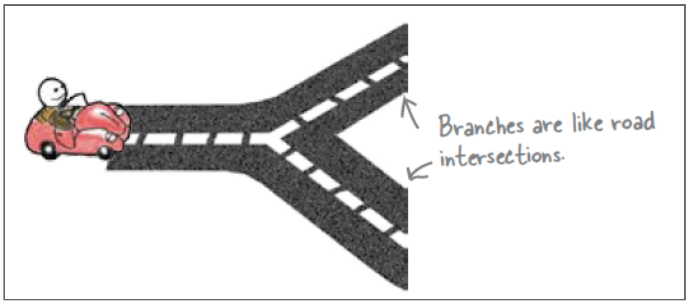
YOUR PROGRAM IS LIKE A NETWORK OF ROADS



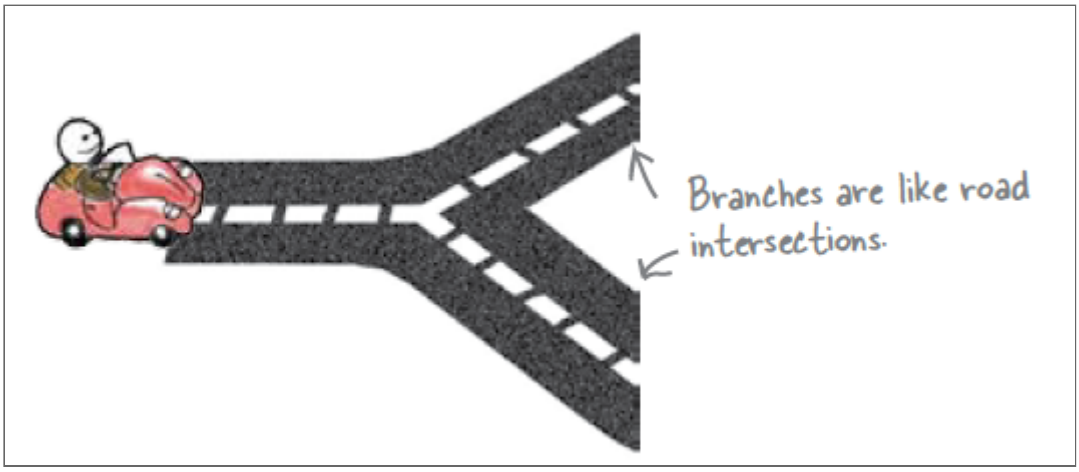
A path refers to the set of instructions that the computer will actually follow (or execute). Your code is like a street network, with lots of sections of code connected together just like the streets in a city. When you drive through a city, you make decisions as to which streets you drive down by turning left or right at different intersections. It's the same for a program. It also needs to make decisions from time to time as to which path to take.

BRANCHES ARE CODE INTERSECTIONS

Driving down a street is easy. You need to make a decision only when you get to an intersection. It's the same for your program. When a program has a list of commands, it can blindly execute them one after another. But sometimes, your program needs to make a decision. Does it run this piece of code or that piece of code?
These decision points are called branches, and they are the road intersections in your code.



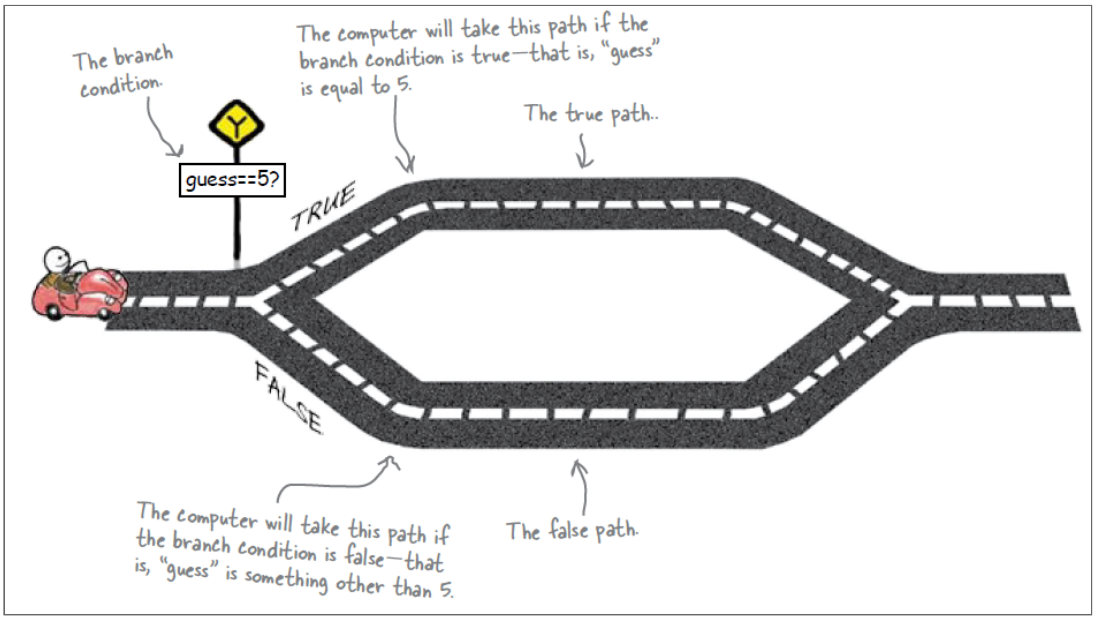
BRANCHES ARE CODE INTERSECTIONS



Your program makes a decision using a branch condition. A branch condition has the value true or false. If the branch condition is true, it runs the code on the true branch. And if the branch condition is false, it runs the code on the false branch.

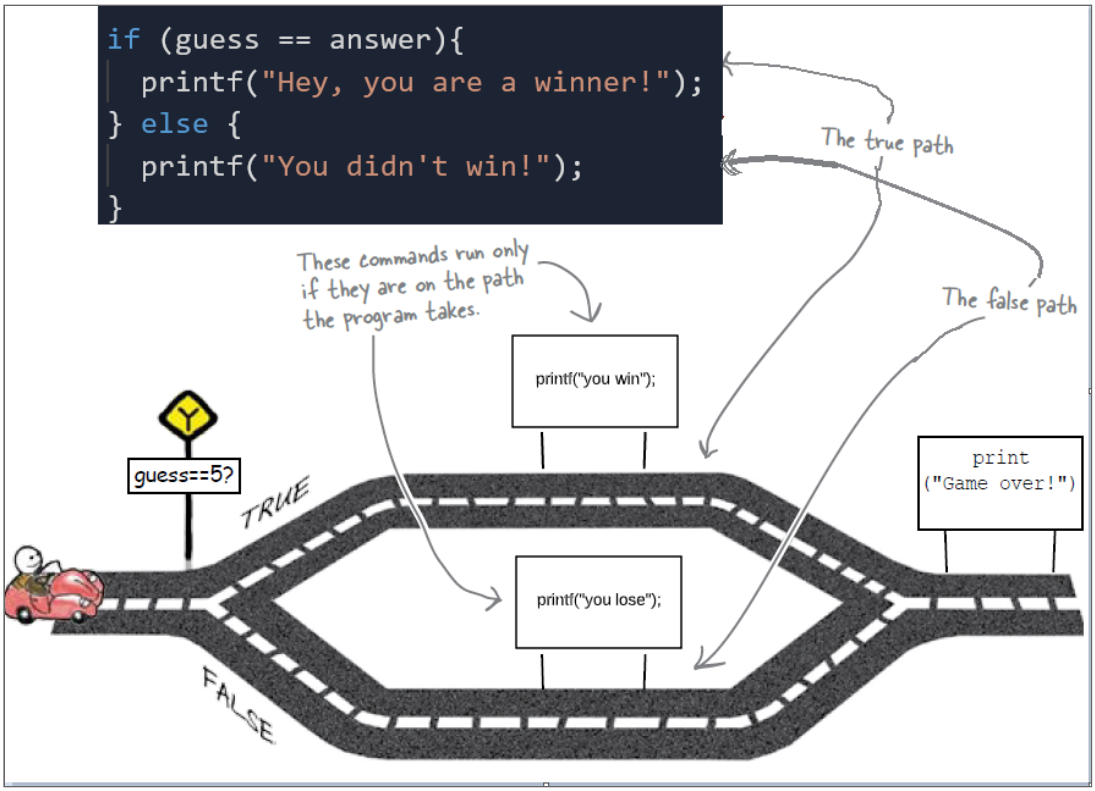
BRANCHES ARE CODE INTERSECTIONS

Your program makes a decision using a branch condition. A branch condition has the value true or false. If the branch condition is true, it runs the code on the true branch. And if the branch condition is false, it runs the code on the false branch.



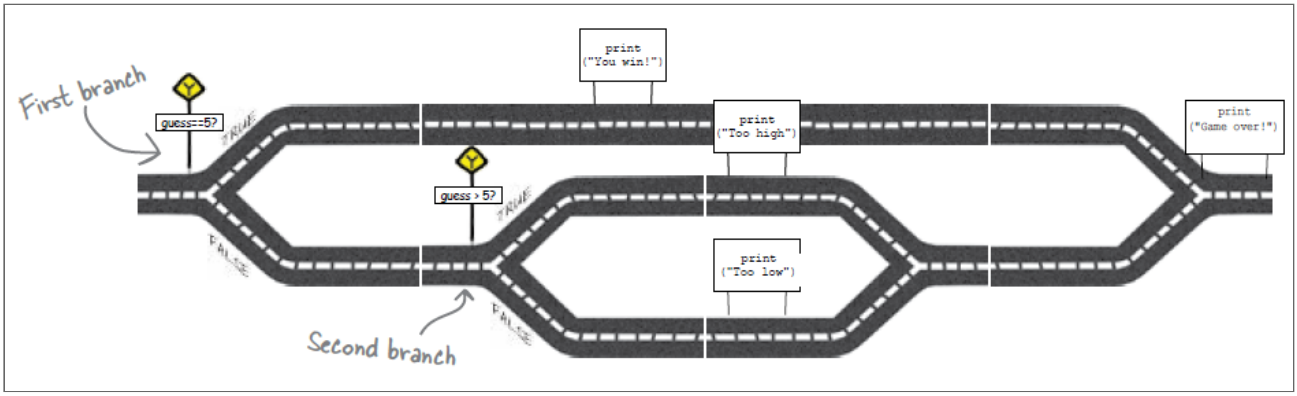
IF/ELSE BRANCHES

We've already seen a branch in the Python game program:

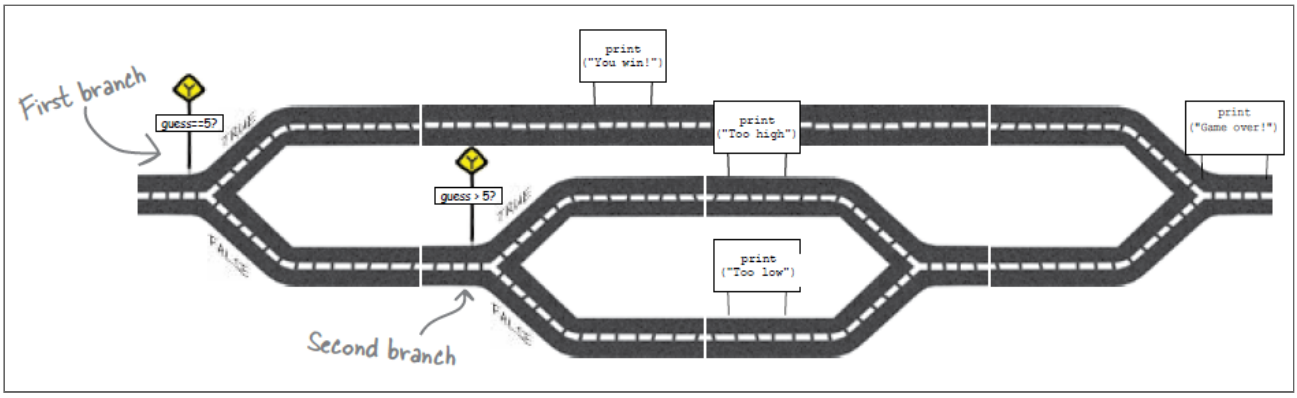


C NEEDS INTERCONNECTING PATHS

The solution's mapped out, and now we know that the program code will need to have paths that match this:



C NEEDS INTERCONNECTING PATHS

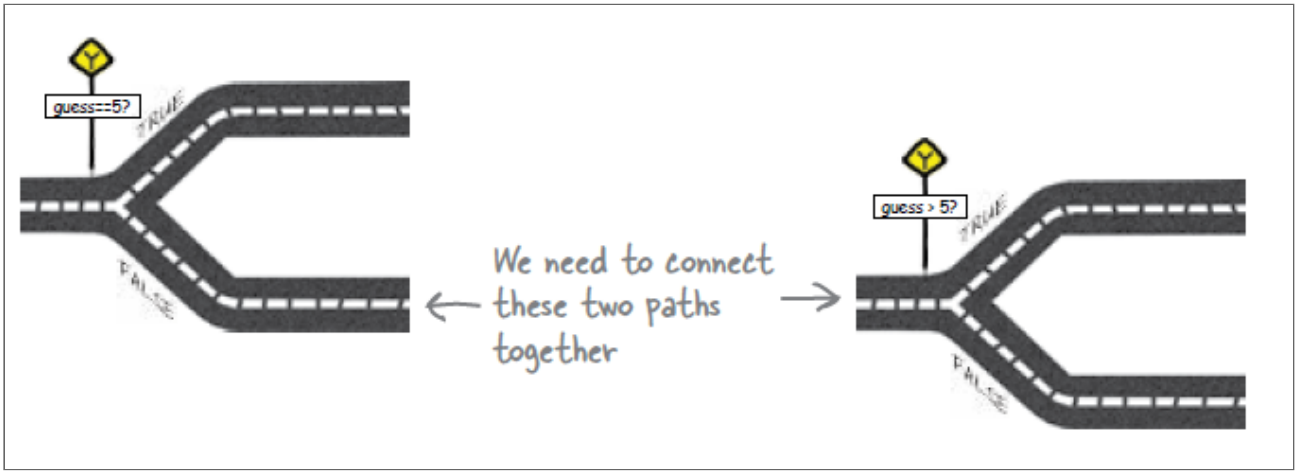


But isn't there a problem here? In the design there are many interconnecting paths, but so far, we have only written code that contains just one branch:

```
btn = digitalRead(BUTTON);
if (btn) {
    digitalWrite(LED_RED, LOW);
    digitalWrite(LED_GREEN, HIGH);
} else {
    digitalWrite(LED_RED, HIGH);
    digitalWrite(LED_GREEN, LOW);
}
```

C NEEDS INTERCONNECTING PATHS

In the new code, we will need to connect two branches together. We need the second branch to appear on the false path of the first.



C USES CURLY BRACES `{}` TO CONNECT PATHS

The code inside the if and else statements is indented and surrounded by 'curly' braces `{}`. While the indentation is only required by convention, the curly braces define a block/branch of code.

```
if (guess == answer){
  printf("Hey, you are a winner!");
} else {
  printf("You didn't win!");
}
```

Speaker notes