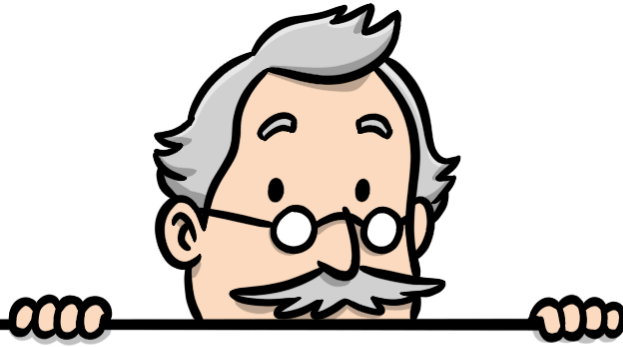




# Illustrated JavaScript coding course

Learn JavaScript building graphical projects



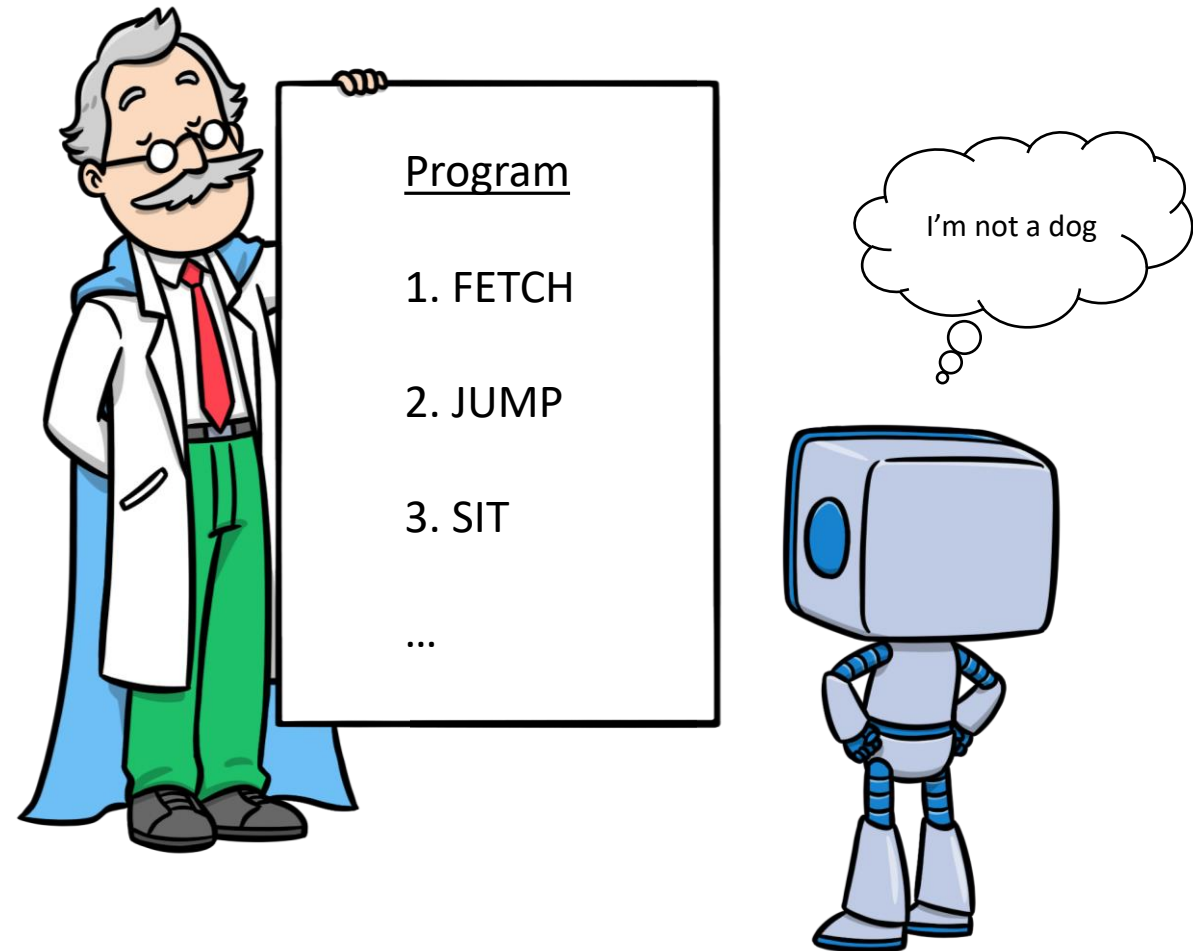
## Chapter I

Welcome to the wonderful world of coding

- What is coding?
- What is JavaScript?
- What is codeguppy.com?
- Creating user accounts

# What is coding?

- **Coding** also known as **computer programming** is the art of giving instructions to your computer (or any other digital device such as your phone, tablet or perhaps your ... robot).
- Your computer is like an obedient dog. If you give it a series of instructions, it will follow them precisely.
- You can even place multiple instructions in a list (aka program) and ask it to execute them all at once.



# Programming languages

Your computer doesn't fetch the ball but can draw a circle on the screen...



- If you want your computer to follow your instructions, you need to write them in a language it understands
- There are literally hundreds of languages (called **programming languages**) that can be used to send instructions to computers
- Each language has its own commands and rules (called **syntax**) that need to be understood in order to write correct programs

# Different kinds of programming languages

- There are hundreds of programming languages available that can be used for all sort of tasks: to build websites, to build games, to build home automations and control robots... and even heat your pizza in the microwave.
- In this course we will learn one of the best languages possible...
- We will learn JavaScript!

JavaScript  
Python  
C++  
Java  
Go  
Julia  
Pascal  
Ada  
Fortran  
BASIC  
C#

## Block-based languages

- Usually used by young kids
- Uses visual interface with drag and drop “commands”
- Good option for small programs
- Some people don’t consider “block-based coding” as real programming

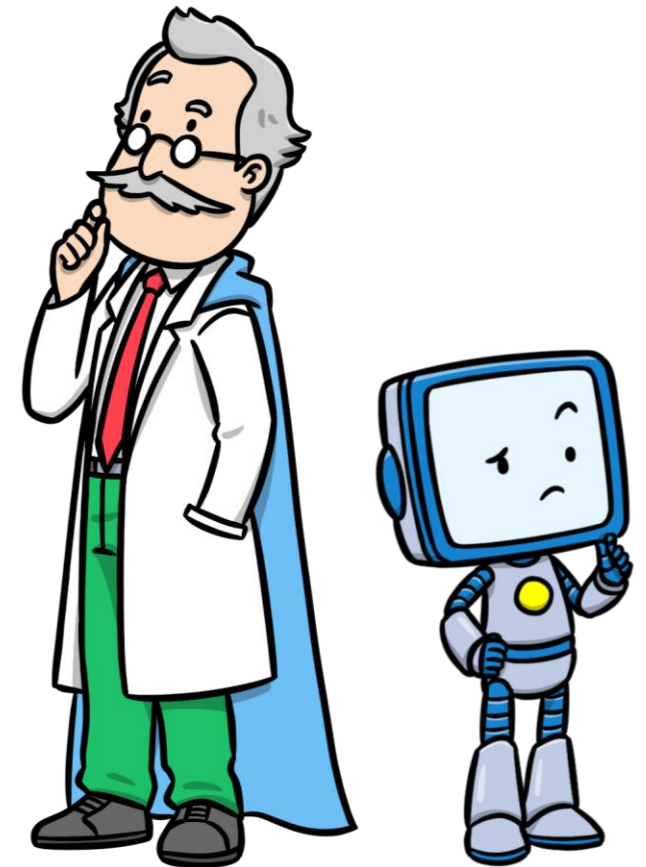
## Text-based languages



- Used by people of all ages
- Require memorization of a small set of “commands”
- Great for building programs of all sizes
- Text-based coding is the “real-coding” that even professionals are using in their day-to-day coding activities

# Why learn JavaScript?

- **JavaScript is by far the most used programming language in the world** - used both by beginners but also by professional software coders.
- People use JavaScript to build websites, create games and nice animations and even program robots. In this course you'll learn how to use JavaScript to **draw with code**, create **animations**, **build games** and other fun projects.
- Coding in a **text-based language** such as JavaScript has also other benefits beyond the world of programming. It enhances problem-solving skills but also attention to details.

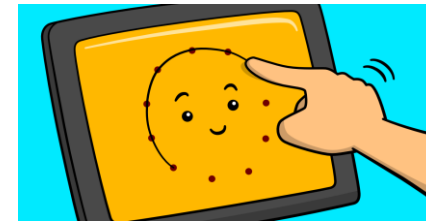
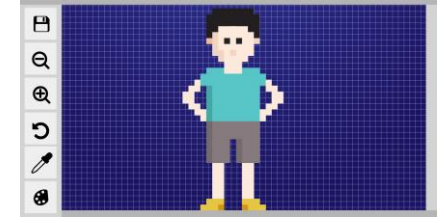


# About codeguppy.com

- codeguppy.com is a **free coding platform** for JavaScript
- you'll write all your **JavaScript** programs directly in a web browser
- you can use any Mac / Windows or Linux PC (some tablets may work as well if you add them a **physical keyboard and mouse**)
- platform comes with **built-in assets** (backgrounds, sprites, etc.) that you can use to build games and other fun projects
- there are **tons of projects** on the platform that you can remix
- Let's start...



codeguppy.com



# Let's open an account with codeguppy.com

- codeguppy.com gives you unlimited space in the cloud to write and store tons of programs
- But first you need to open an account with codeguppy.com
- You need to provide a valid email address and a password (website doesn't ask any other personal details)

Note: After registration, you should get an email from codeguppy.com. Please verify your email by clicking the link in the email.

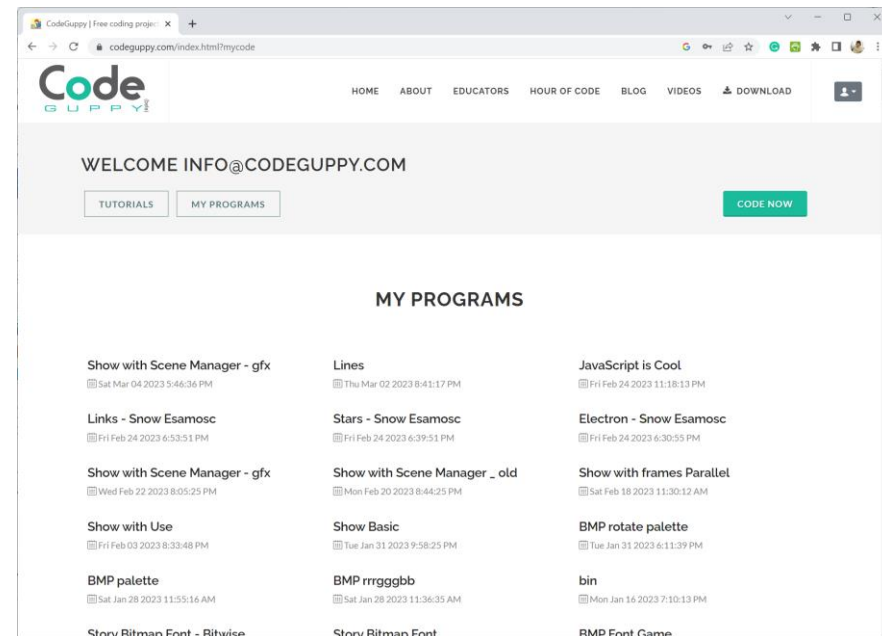
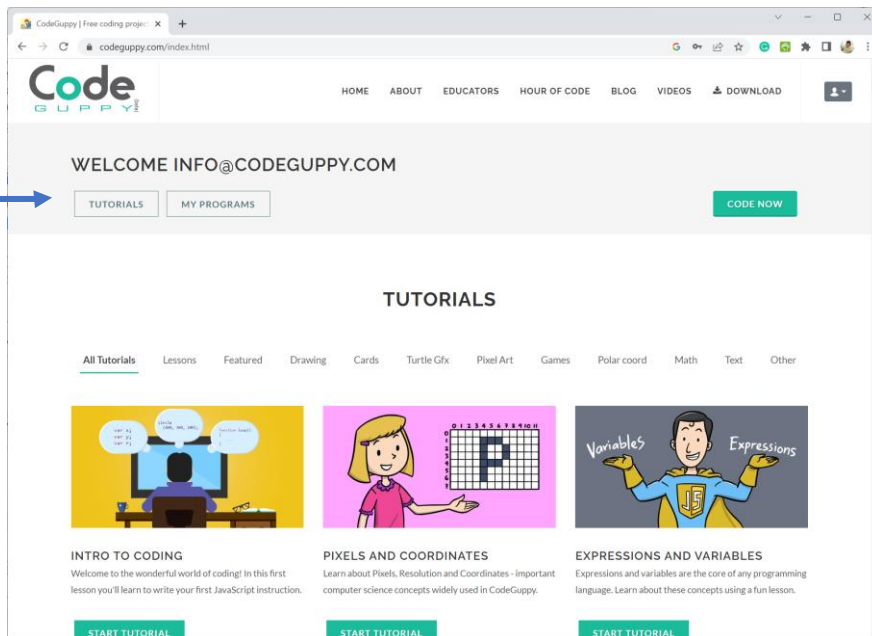
The image shows a registration form for codeguppy.com. At the top, there is a logo of a laptop with a globe icon and the text "codeguppy.com". Below the logo, there is a green button labeled "JOIN FOR FREE". A blue line with circular markers numbered 1, 2, and 3 indicates the registration process. Step 1 points to the "JOIN FOR FREE" button. Step 2 points to the "Register with email" section, which includes an "Email:" label, an empty text input field, a "Password:" label, and another empty text input field. Step 3 points to a dark blue button labeled "REGISTER NOW" at the bottom right of the form.



# Main Page

After you completed the registration, and verification of your email address, you should see a banner on your home page with three buttons.

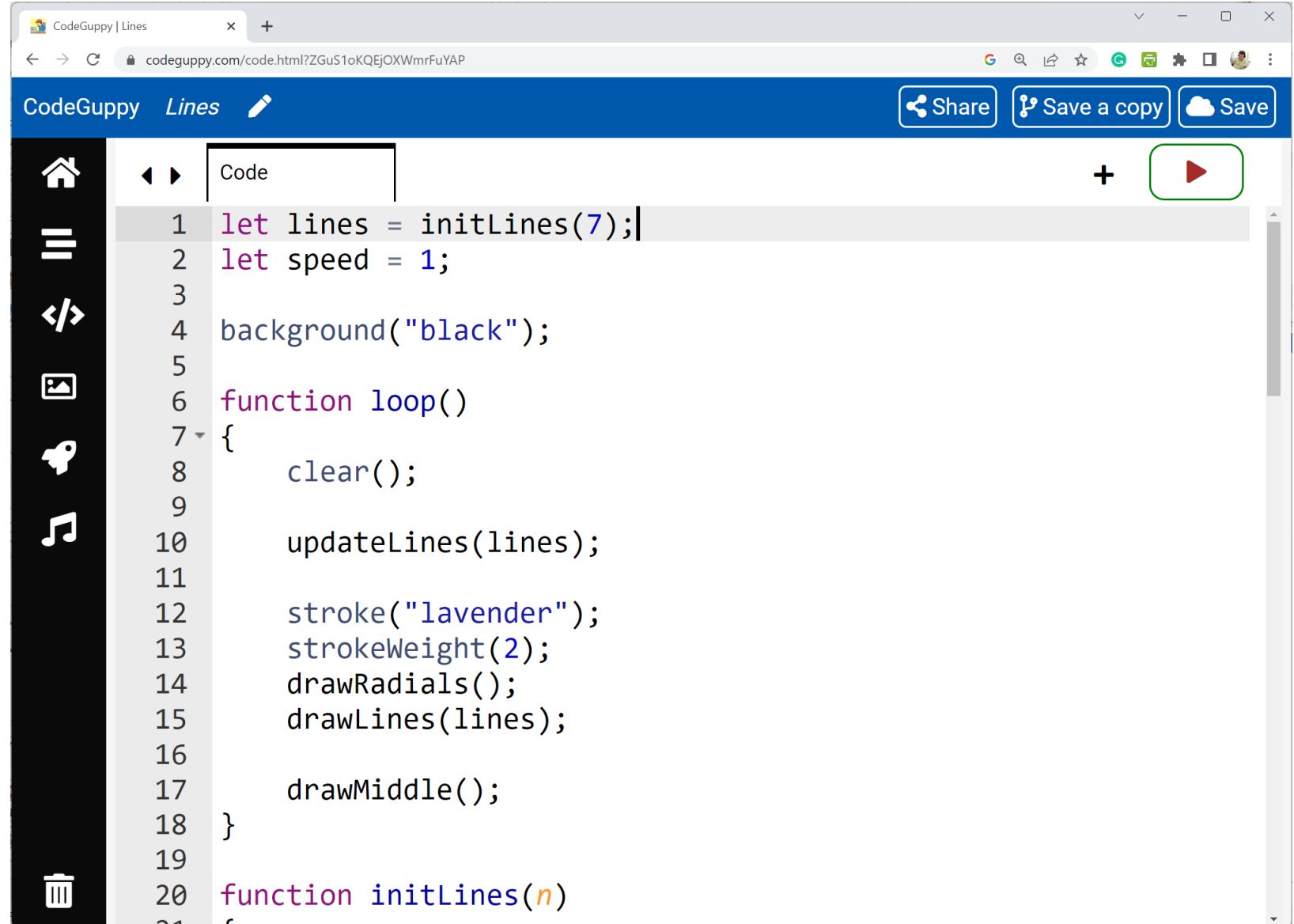
- **TUTORIALS** will show you the built-in projects and tutorials available on the website. You'll surely learn a lot inspecting these!
- **MY PROGRAMS** will show you the list of the programs you created. At the beginning this list will be empty but will grow in time.
- **CODE NOW** is the button that will use to launch an empty code editor in order to create a new program



# Code Editor

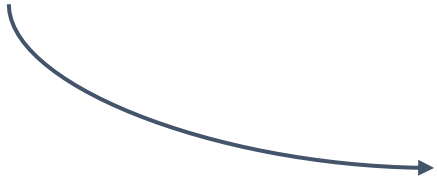
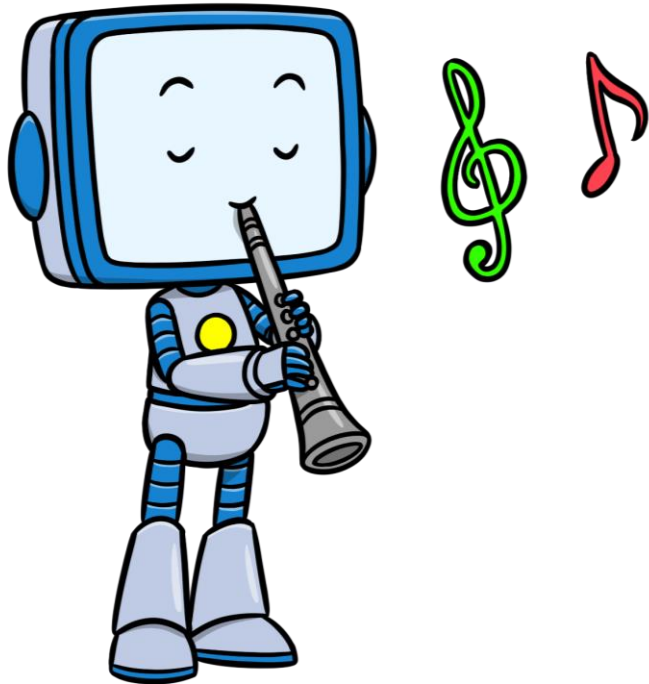
This is the code editor. We will use it a lot in our coding explorations.

For now, remember that if you open it by mistake, you can always close it and return to the home page by using the home button:

A screenshot of the CodeGuppy web-based code editor. The browser address bar shows the URL 'codeguppy.com/code.html?ZGuS1oKQEjOXWmrFuYAP'. The editor has a blue header with 'CodeGuppy Lines' and buttons for 'Share', 'Save a copy', and 'Save'. A left sidebar contains icons for home, menu, code, image, push, music, and trash. The main area shows a code editor with a 'Code' tab and a play button. The code is as follows:

```
1 let lines = initLines(7);|
2 let speed = 1;
3
4 background("black");
5
6 function loop()
7 {
8   clear();
9
10  updateLines(lines);
11
12  stroke("lavender");
13  strokeWeight(2);
14  drawRadials();
15  drawLines(lines);
16
17  drawMiddle();
18 }
19
20 function initLines(n)
```


Feel free to explore the editor...




### Sprites

Characters Decor Items Sets Tiles


adventure girl holidays boy cat animals dog dino vehicles




adventure\_girl




santa




knight



ninja



cute\_girl



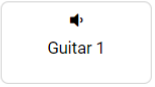
cat

Drag items to code area...

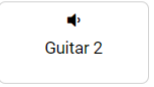
### Music and Sounds

Music Sounds

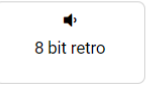
guitar retro misc ambience



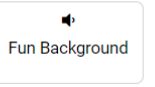
Guitar 1



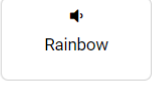
Guitar 2



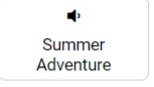
8 bit retro



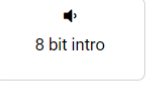
Fun Background



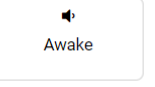
Rainbow



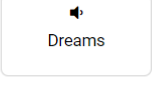
Summer Adventure



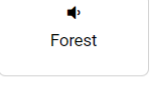
8 bit intro



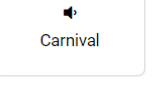
Awake



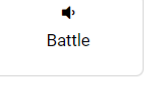
Dreams



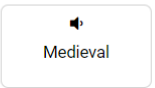
Forest



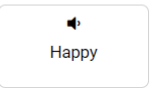
Carnival



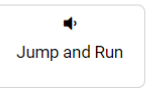
Battle



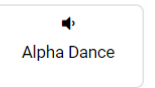
Medieval



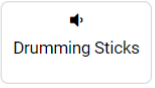
Happy



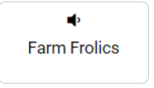
Jump and Run



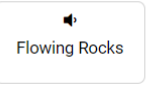
Alpha Dance



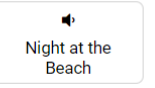
Drumming Sticks



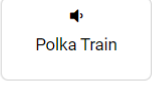
Farm Frolics



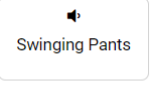
Flowing Rocks



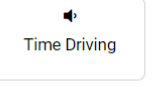
Night at the Beach



Polka Train



Swinging Pants



Time Driving

Drag items to code area...



## Chapter II

# Let's write some code

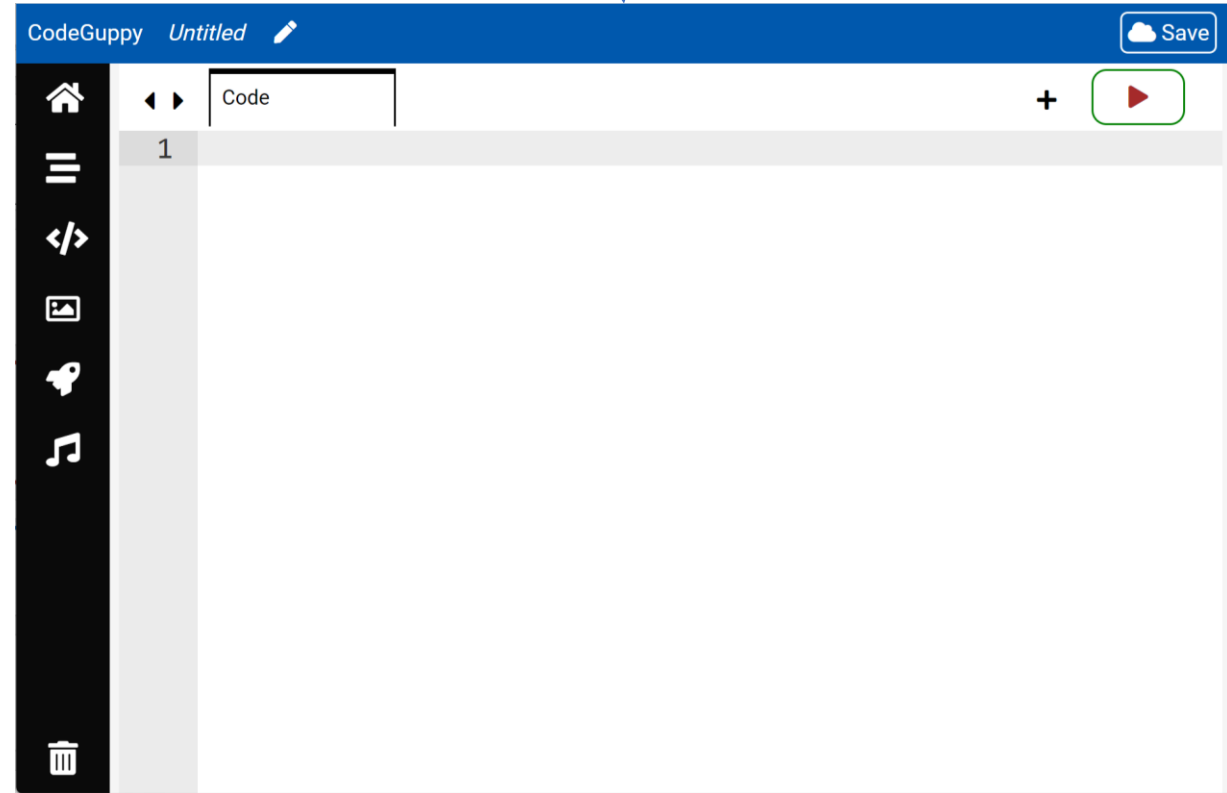
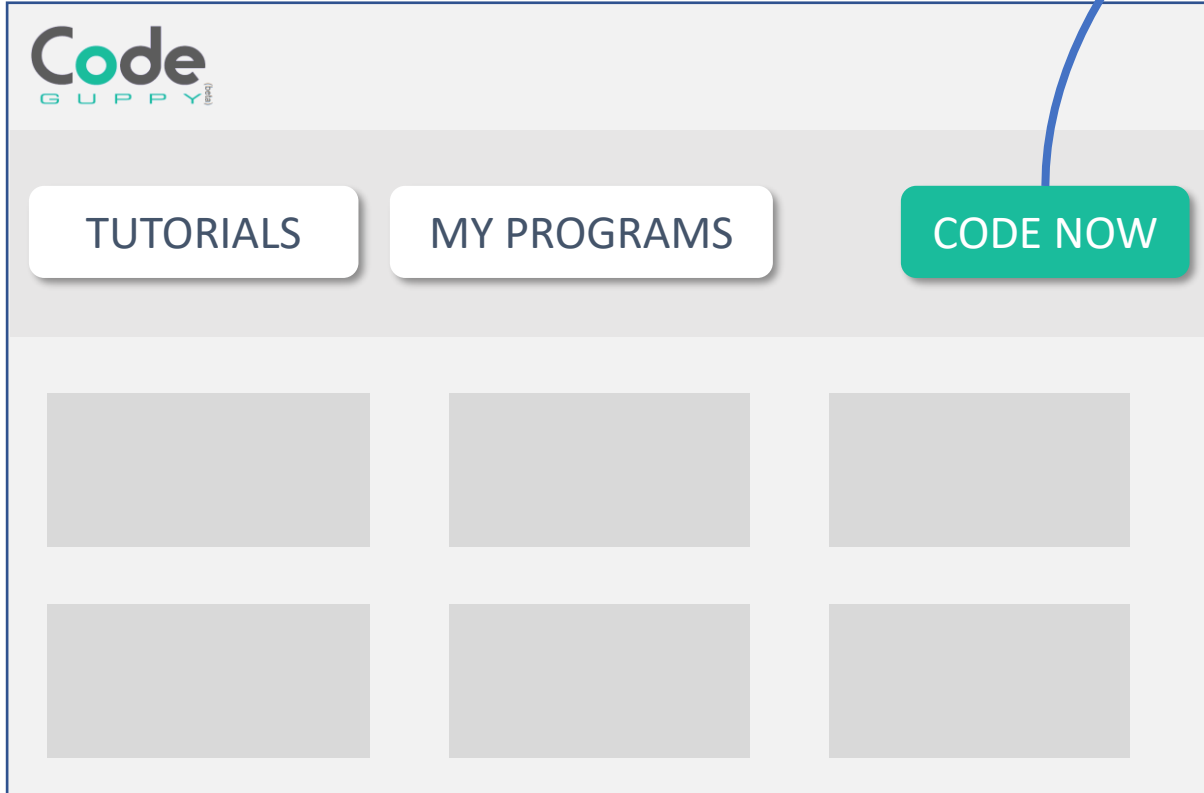
Exploring the code editor

Build a program using drag and drop

Our first type-in program

Homework

# Launching the code editor



### Run / Stop Button

Run or Stop your program.

### Action Bar

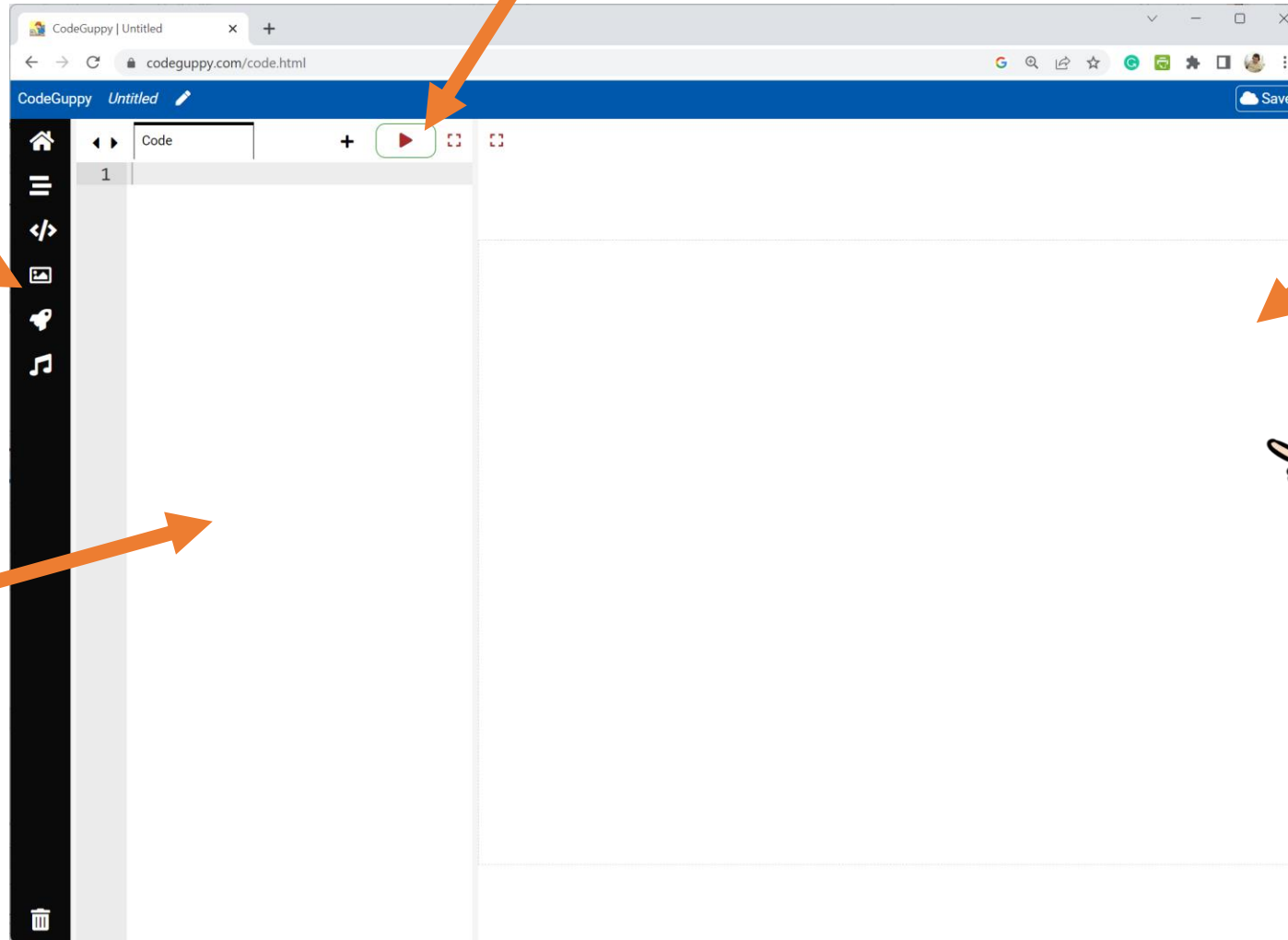
Browse sprite library, colors, commands, etc.

### Code Editor

Here you'll type your programs

### Output Canvas

Here you'll see the output of your program.

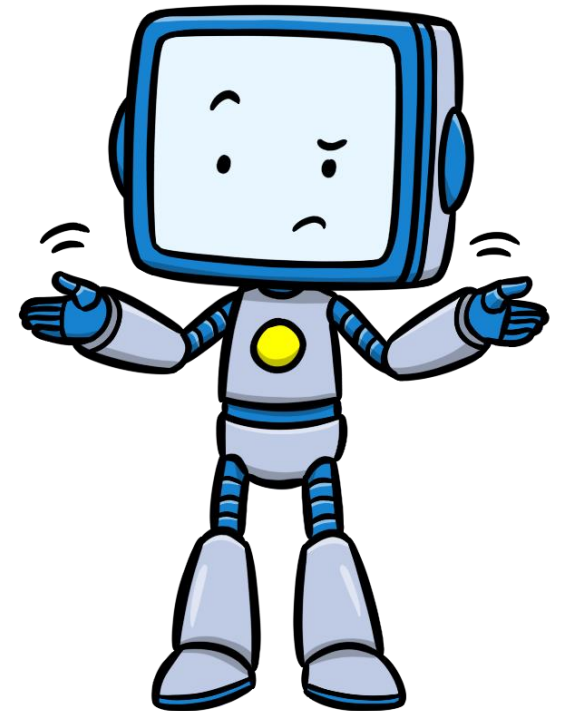


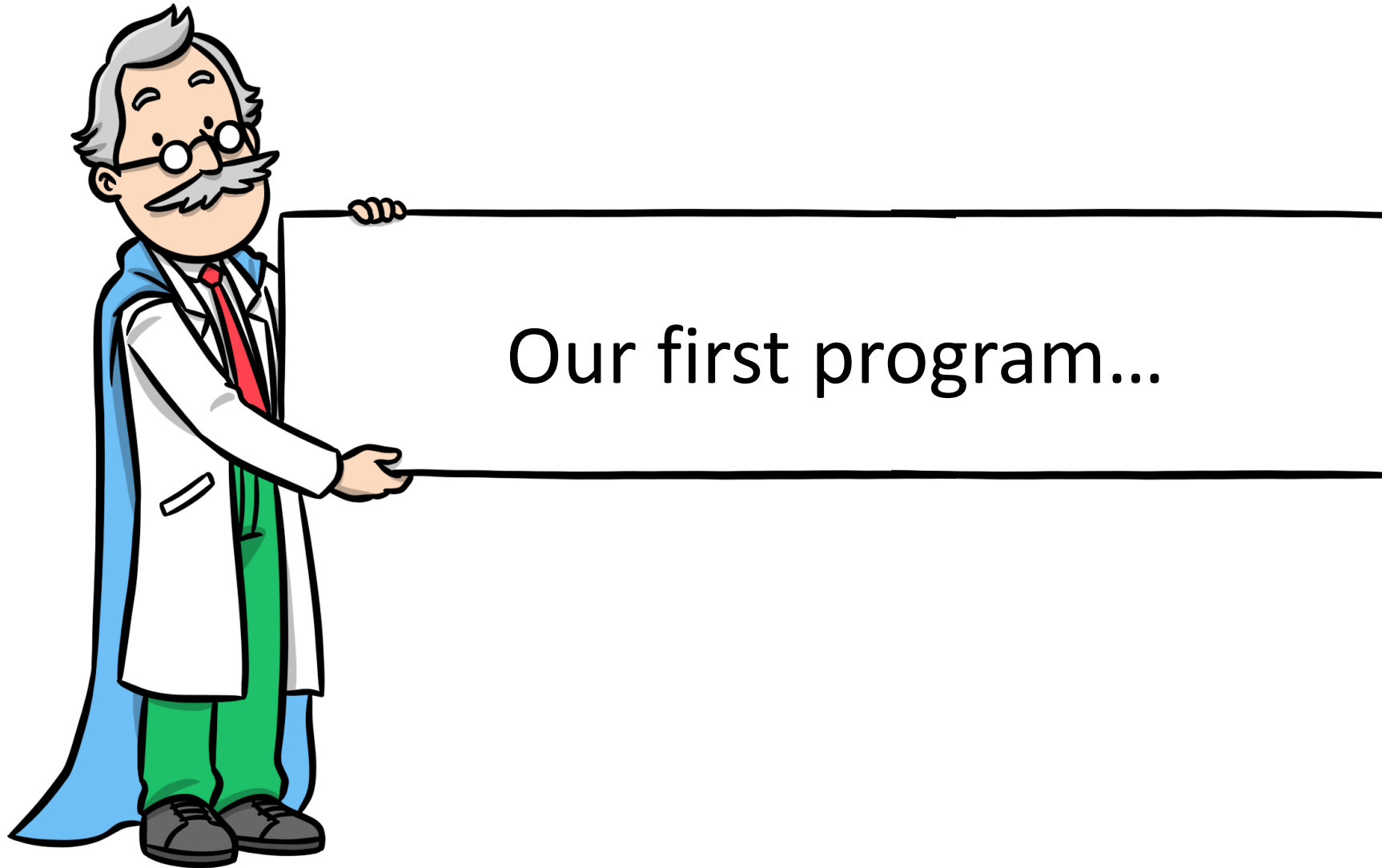
If you have a high screen resolution, your screen will be split in half: on the left you'll see the code editor and on the right the output area

Programming is like  
writing a book...

... except if you miss out a  
single comma on page 349  
the whole thing you wrote  
makes no sense.

- Anonymous





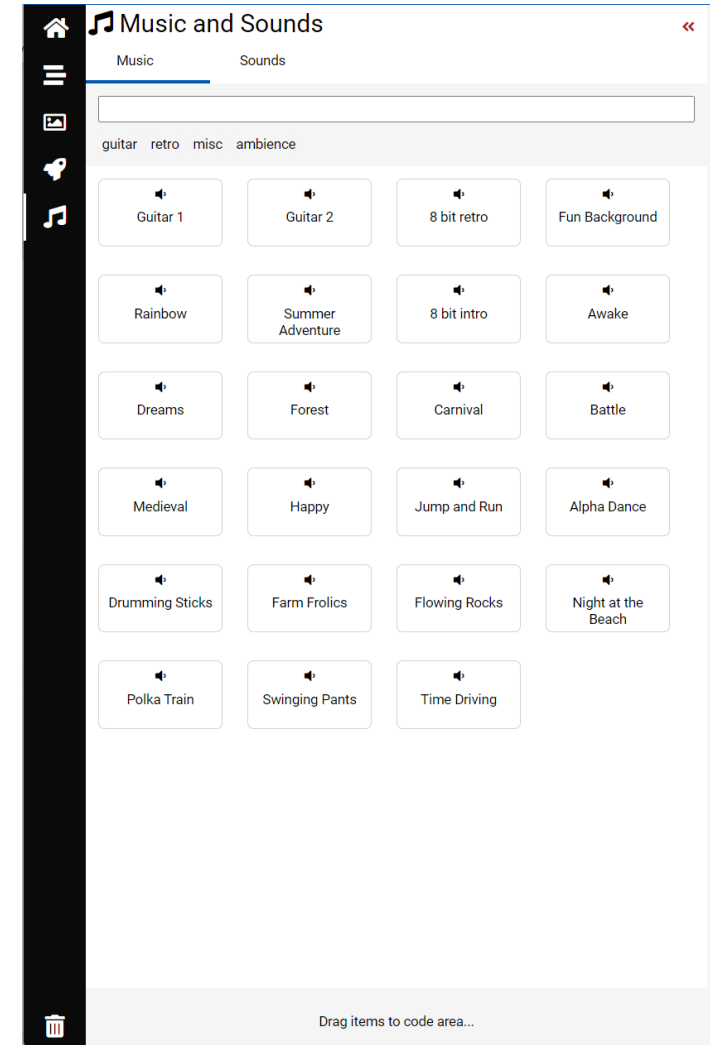
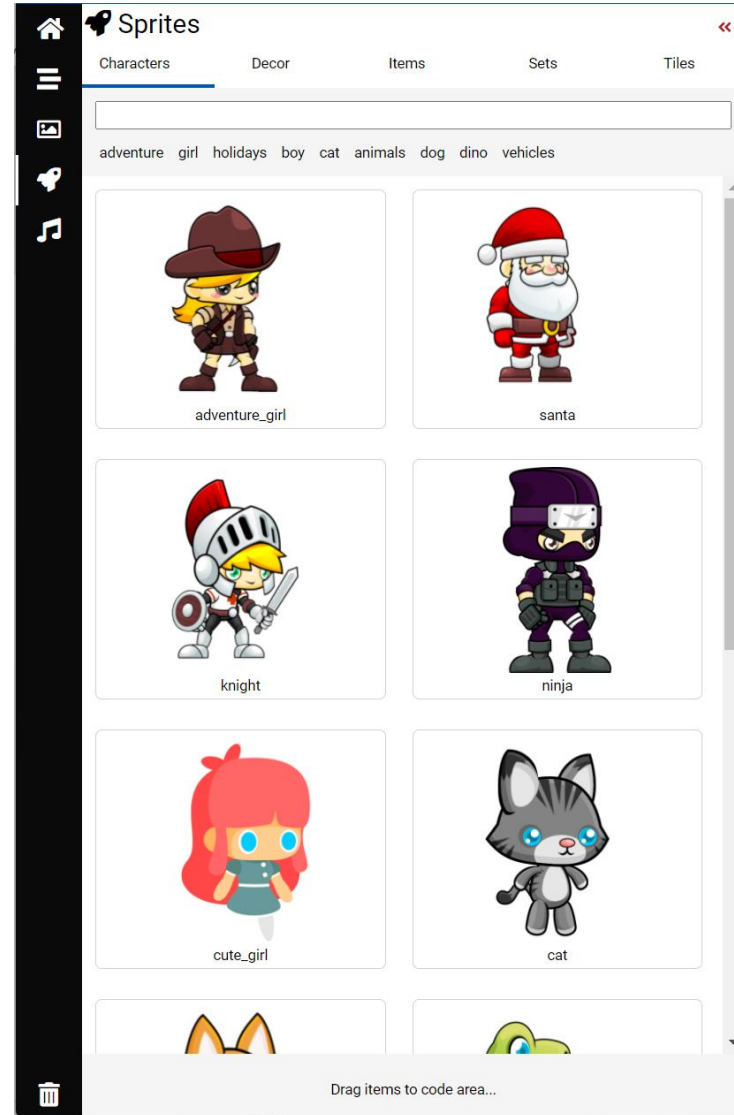
Our first program...



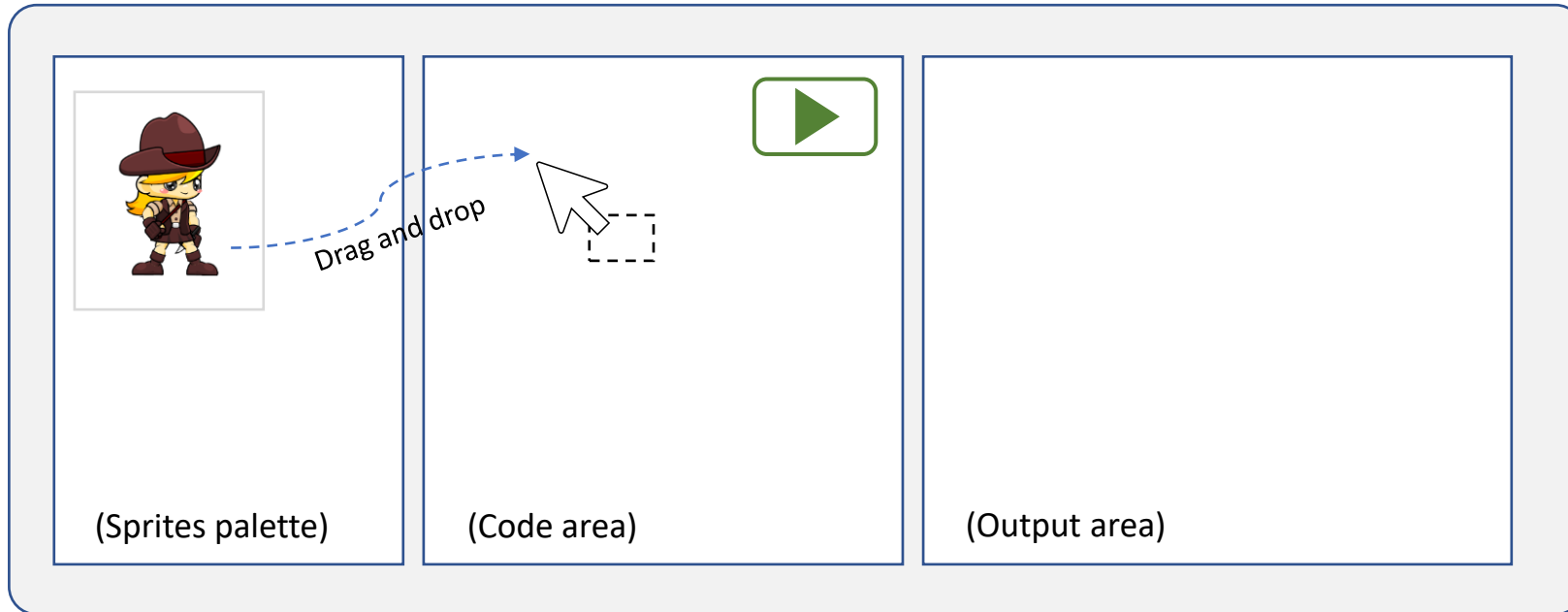
# Let's play with built-in assets

The buttons on the action bar show and hide various *pallets* with instructions, commands or assets.

For now, let's drag and drop a sprite in an empty area of the code editor...



# Drag and drop a sprite



The platform will write a small “line of code” after you drop a sprite in the code area, like this:

```
sprite('adventure_girl.idle', 400, 300, 0.5);
```



# Each sprite outputs a different line of code...



```
sprite('adventure_girl.idle', 400, 300, 0.5);
```




```
sprite('knight.idle', 400, 300, 0.5);
```

You don't have to write the above code. Just drag and drop a sprite from the palette in the code editor to have this code generated for you.

# Let's test our code...

```
sprite('plane.fly', 400, 300, 0.5);
```



- Press “Run” to execute the program 



- If you don't have any error in your program, you should see in the output area the sprite you dropped in the code!

If your program doesn't display anything, delete the code in the editor and try drag and dropping another sprite. Then press “Run” again.

# Stopping the program

```
sprite('plane.fly', 400, 300, 0.5);
```



- After execution, the program will run until you stop it.
- If you want to modify your program, you first need to Stop it first, then modify the code (we'll see later how to do this) and then Run it again. It is a continuous cycle of: Edit -> Run -> Stop -> Edit -> Run -> Stop
- To stop the program, use the "Stop" button  or the close output button 



Let's analyze the code...

```
sprite('adventure_girl.idle', 400, 300, 0.5);
```

Built-in instruction that asks computer to display a sprite on the canvas.

The actual sprite is specified in between quotes inside the parenthesis as a "parameter".

Parameters of the instruction.

Parameters specify what sprite to display, where to display it on the canvas and at what coordinates.

**Experiment:** Try to change the 400 and 300 numbers with other numbers between 0 and 800.



Press "Play" / "Stop" after each modification to run / stop the program.

A diagram illustrating the drag-and-drop method for creating a JavaScript program. It shows a sequence of three panels: 1. A green airplane with a pilot in a red helmet is shown in a white box. 2. A dashed blue arrow labeled "Drag and drop" points from the airplane to a code editor. The code editor contains the text `sprite('plane.fly', 400, 300, 0.5);` and a play button icon. 3. The airplane is shown flying in the final panel.

Congratulations!

You just created your first one-line long JavaScript program using the drag-and-drop method!

# Let's build now a greeting card...

**Step 1:** Drag and drop a music file

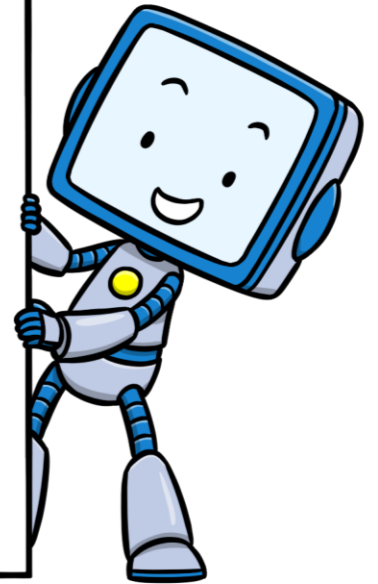
**Step 2:** Drag and drop a background image or color

**Step 3:** Drag and drop a sprite

```
music('Fun Background');  
background('Field');  
sprite('plane.fly', 400, 300, 0.5);
```

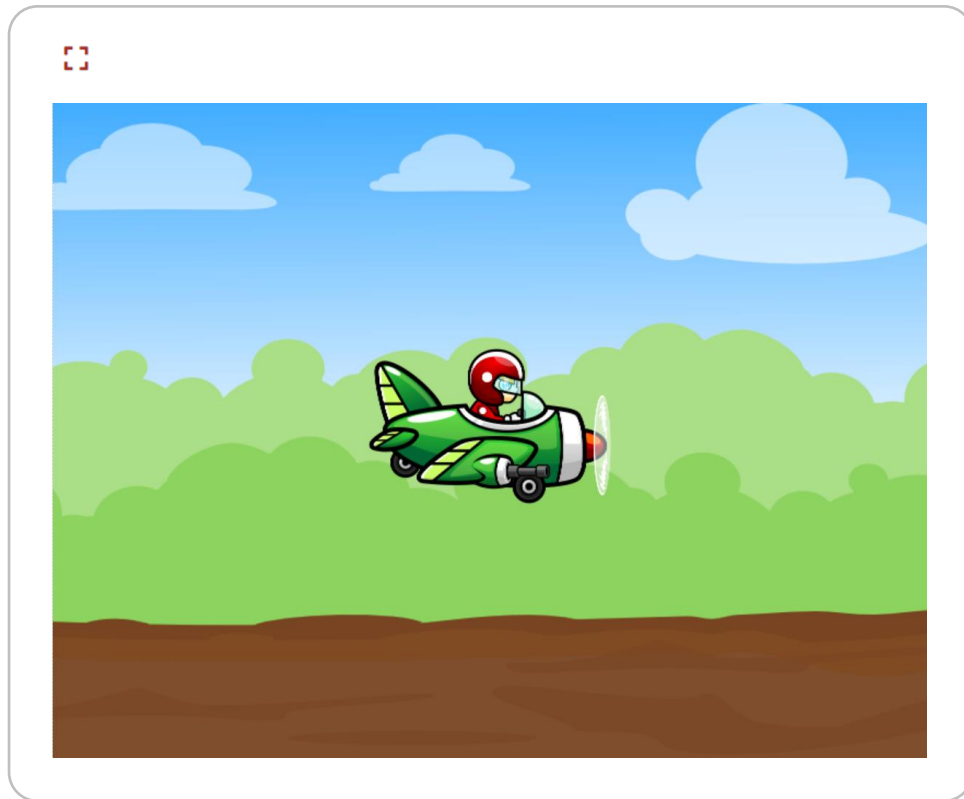


Press "Play" when ready.





- With only 3 lines of code, you can create quite interesting greeting cards
- Feel free to explore different combinations of backgrounds, sprites and music
- Do you know how to adjust the sprite position to better fit in your composition?



**Example 1**

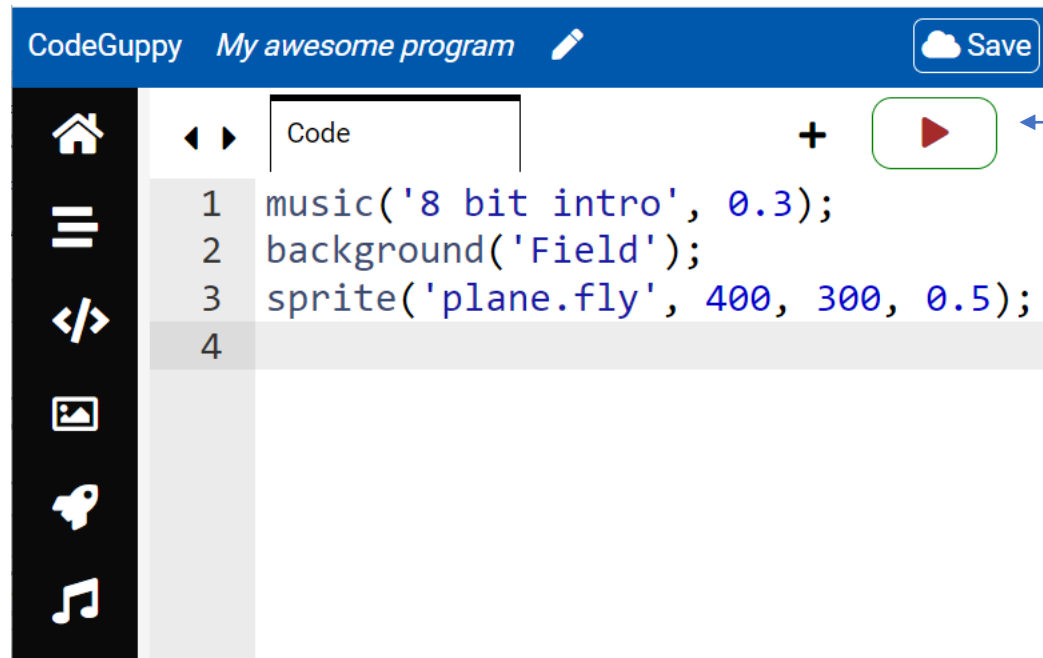


**Example 2**

# Naming and saving the program

Click on the Pencil button or inside the label to edit the name of your program

Then click on the Save button to save your code



# Sharing programs...



- When a program is saved, a new “Share” button appears on the toolbar.
- Use the link to share the program with family and friends (via email)
- You may also submit homework and assignments in the same way using the classroom system



### Share Program ×

Share the source code together with your program

Link to share this program  
 Copy

Embed your program to your blog or website by copying the following code  

```
<iframe src="https://www.codeguppy.com/code.html?DQuO2ABz8Obtn5T8vL07" width=1024 height=670></iframe>
```

Copy

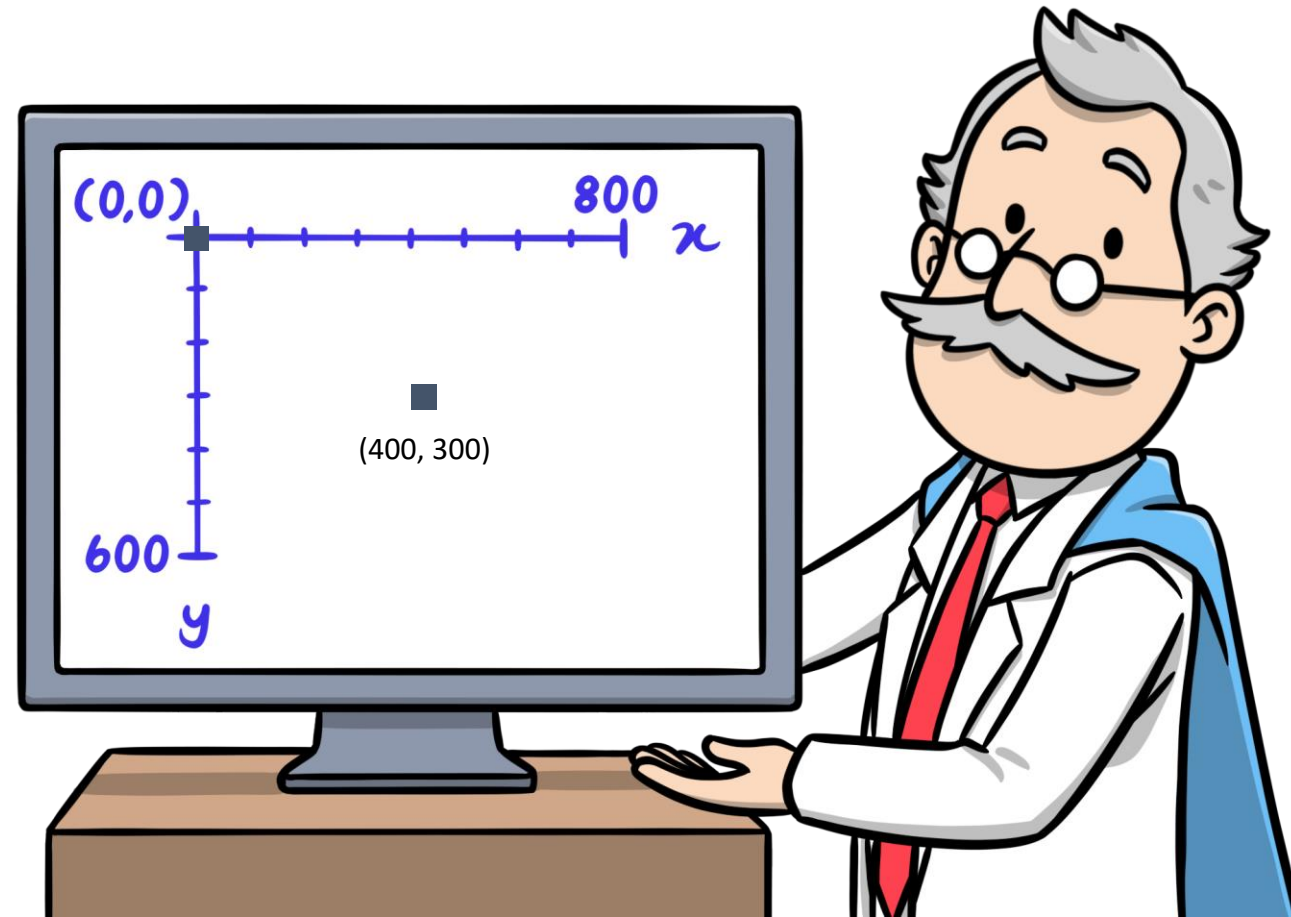
**COOL!**



Our first type-in program...

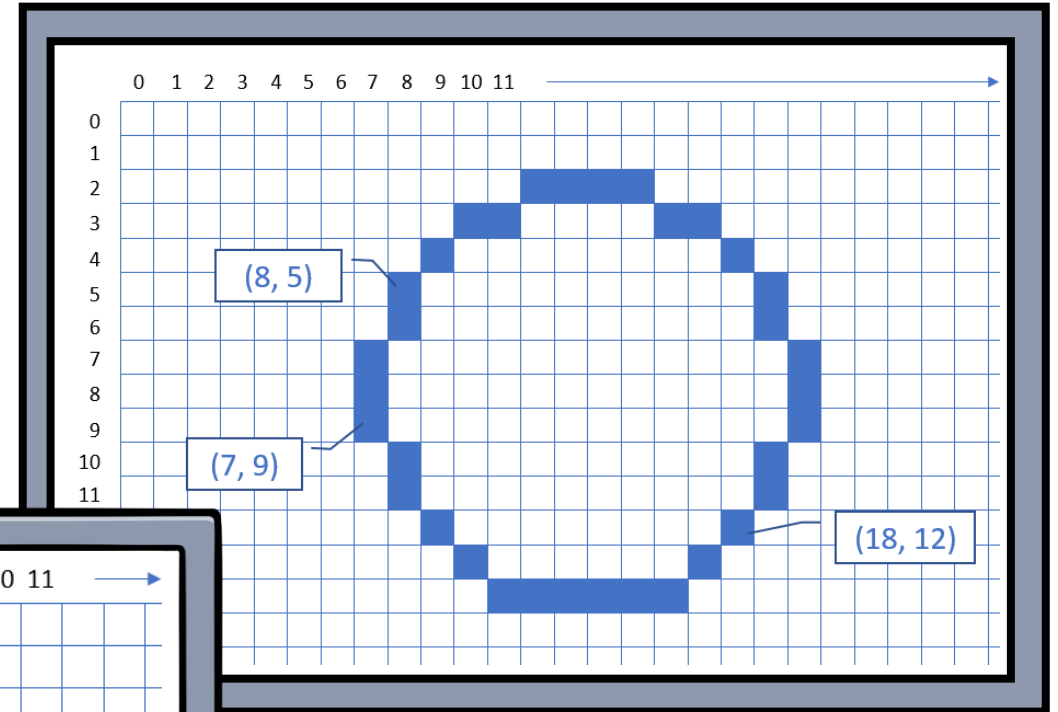
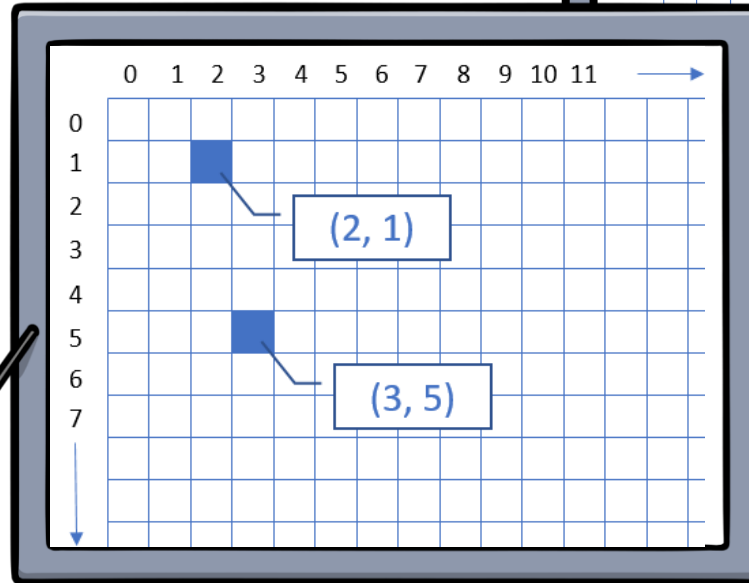
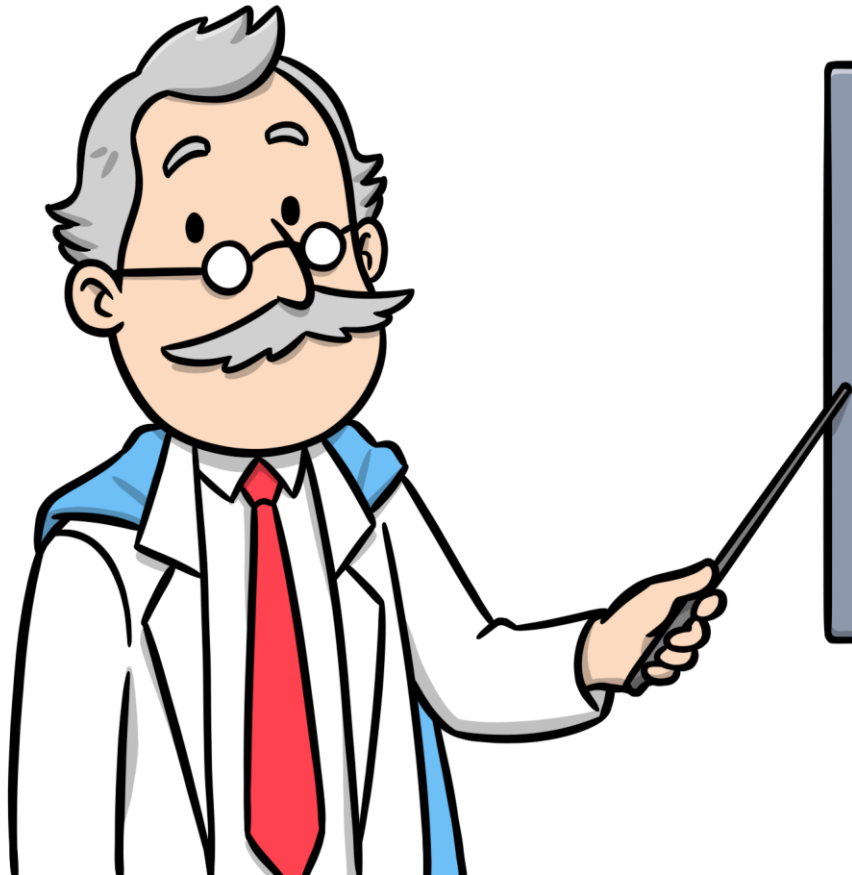
# Let's understand the canvas

- In codeguppy.com, programs can write and draw on a graphical canvas of 800x600 pixels
- Origin is in the top-left corner
- Middle of the canvas is at about (400, 300)
- x coordinate goes from 0 to 800 (left to right)
- y coordinate goes from 0 to 600 (top to bottom)



# Pixels and Coordinates

- Canvas is made from many tiny square dots called **pixels** (480,000 pixels: 800 on horizontal x 600 on vertical)
- **Coordinates** are a pair of (x, y) numbers which are used to determine the position of a pixel on the canvas.



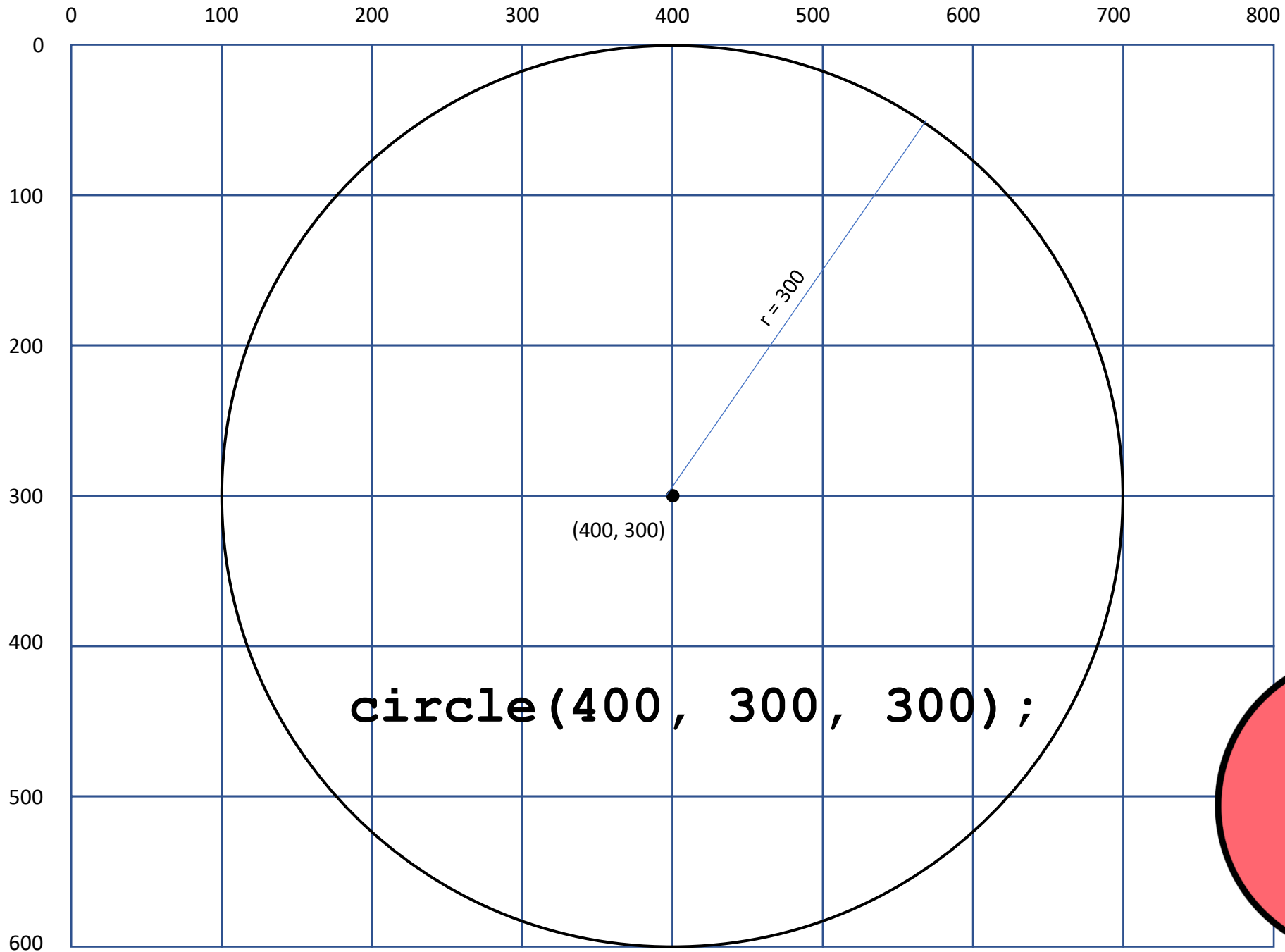
# Our first type-in program



```
circle(400, 300, 300);
```

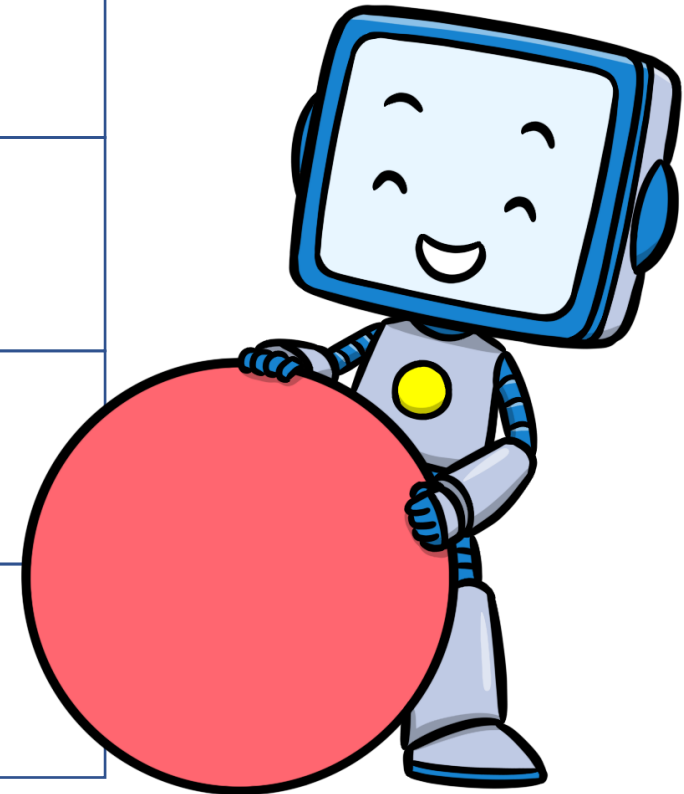


- Type **carefully** this line in the code editor.
- Make sure you use the same  **casing**  as illustrated and include all the  **punctuation**  signs.
- When ready, press the  **Play / Run**  button



This is the  
output:

A big circle!







Built-in instruction that asks computer to draw a circle on the canvas.

codeguppy.com has many built-in instructions for different purposes (remember the `sprite` instruction used before).

Parameters of the instruction.

There are 3 parameters inside **parenthesis** and separated by **comma**:

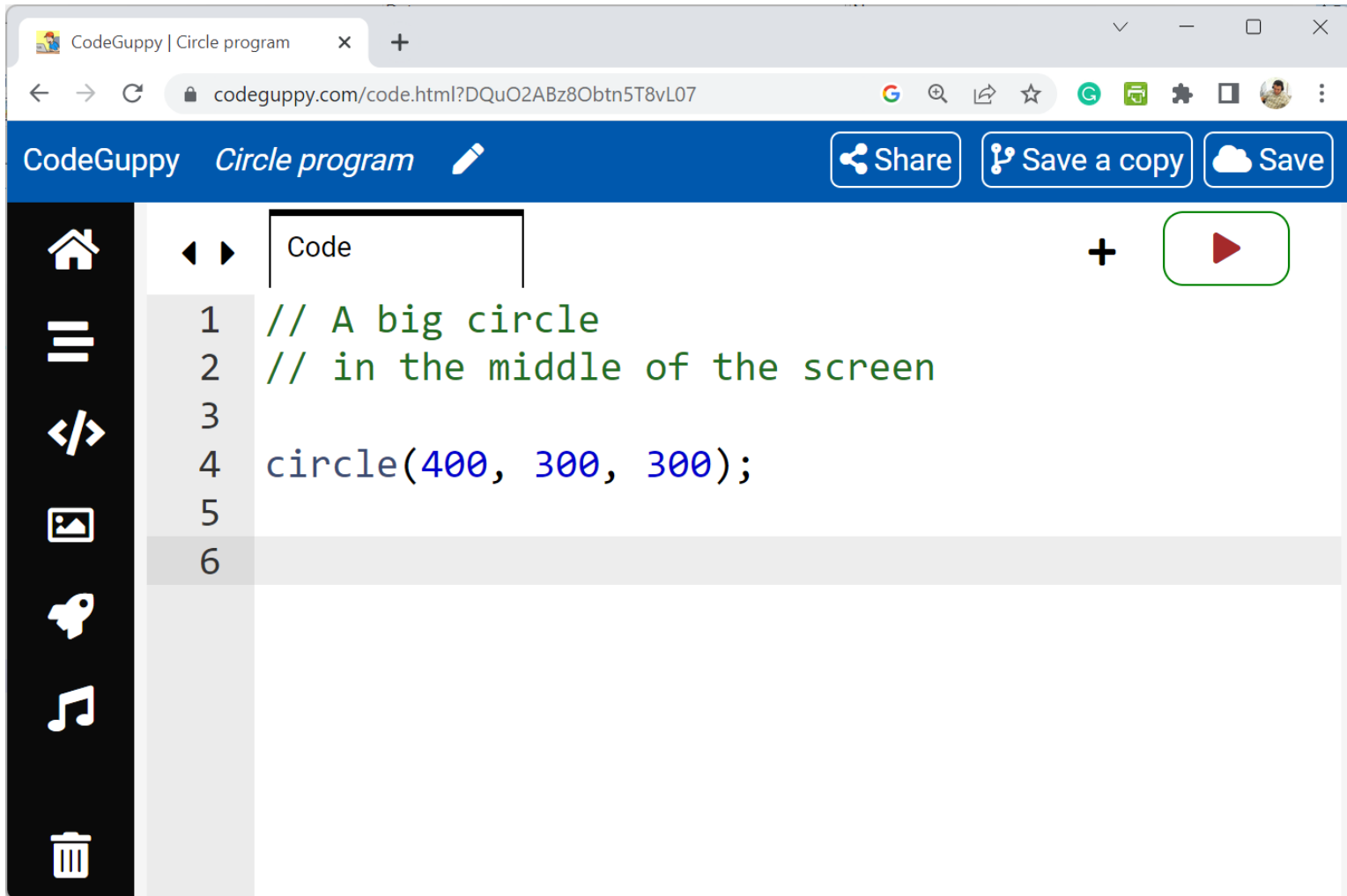
- 400, 300 - coordinates of circle center
- 300 – radius of the circle



Try to modify the parameters of this instruction and notice the effect.

Don't forget to press "Play" / "Stop" after each modification.

# Write readable code



The screenshot shows a web browser window with the CodeGuppy interface. The address bar shows the URL `codeguppy.com/code.html?DQuO2ABz8Obtn5T8vL07`. The page title is "CodeGuppy | Circle program". The main content area displays a code editor with the following code:

```
1 // A big circle
2 // in the middle of the screen
3
4 circle(400, 300, 300);
5
6
```

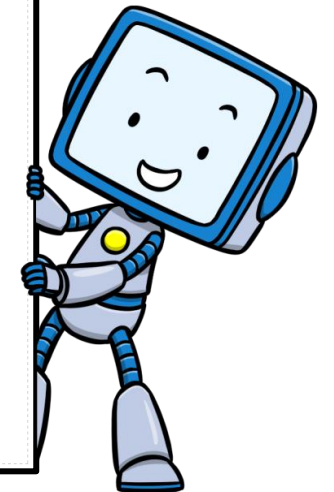
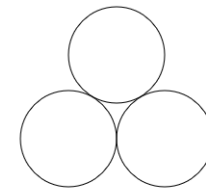
The code editor has a dark background and a light-colored text. The code is written in a monospace font. The first two lines are comments, starting with `//`. The third line is empty. The fourth line is the function call `circle(400, 300, 300);`. The fifth and sixth lines are empty. The code editor has a toolbar with a play button and a plus sign. The browser window has a navigation bar with "Share", "Save a copy", and "Save" buttons.

- Whitespaces don't matter in JavaScript. Use "spaces" inside your program to make it look nice. For instance, you can place a "space" after each parameter.
- You can also write "comments" inside your code. If you start a line with `//` you can write whatever you want on that line – the computer will ignore it.
- Even if the computer ignores a comment line, comments are great for people to remember what a particular program or line of code is doing.

# How many circles can you draw?

- Go ahead and play with the circle instruction. Draw circles in various positions on the screen
- Add multiple lines to your program
- For instance, this small program draws three circles

```
circle(400, 263, 50);  
circle(350, 350, 50);  
circle(450, 350, 50);
```



Let's draw a bear using circles...



Type carefully the program that you see in the listing

```
// Draw bear face
circle(400, 300, 200);

// Draw left ear
circle(250, 100, 50);
circle(270, 122, 20);

// Draw right ear
circle(550, 100, 50);
circle(530, 122, 20);

// Draw left eye
circle(300, 220, 30);
circle(315, 230, 10);

// Draw right eye
circle(500, 220, 30);
circle(485, 230, 10);

// Draw nose
circle(400, 400, 90);
circle(400, 350, 20);
```

Here are a few tips to ensure programs are type-in correctly:

- Please type very carefully exactly as it appears on the slide. Do not skip any letter, number or punctuation sign!
- Programs are case sensitive. Type-in all the commands using the same case as you see on the slide.
- To avoid accumulating errors, you can run the program from time to time, but only after the current line of code is completed.
- If computer will output errors, please check carefully the typed-in program against the program on the slide. Ask for assistance if you need help.

```
// Draw bear face  
circle(400, 300, 200);
```

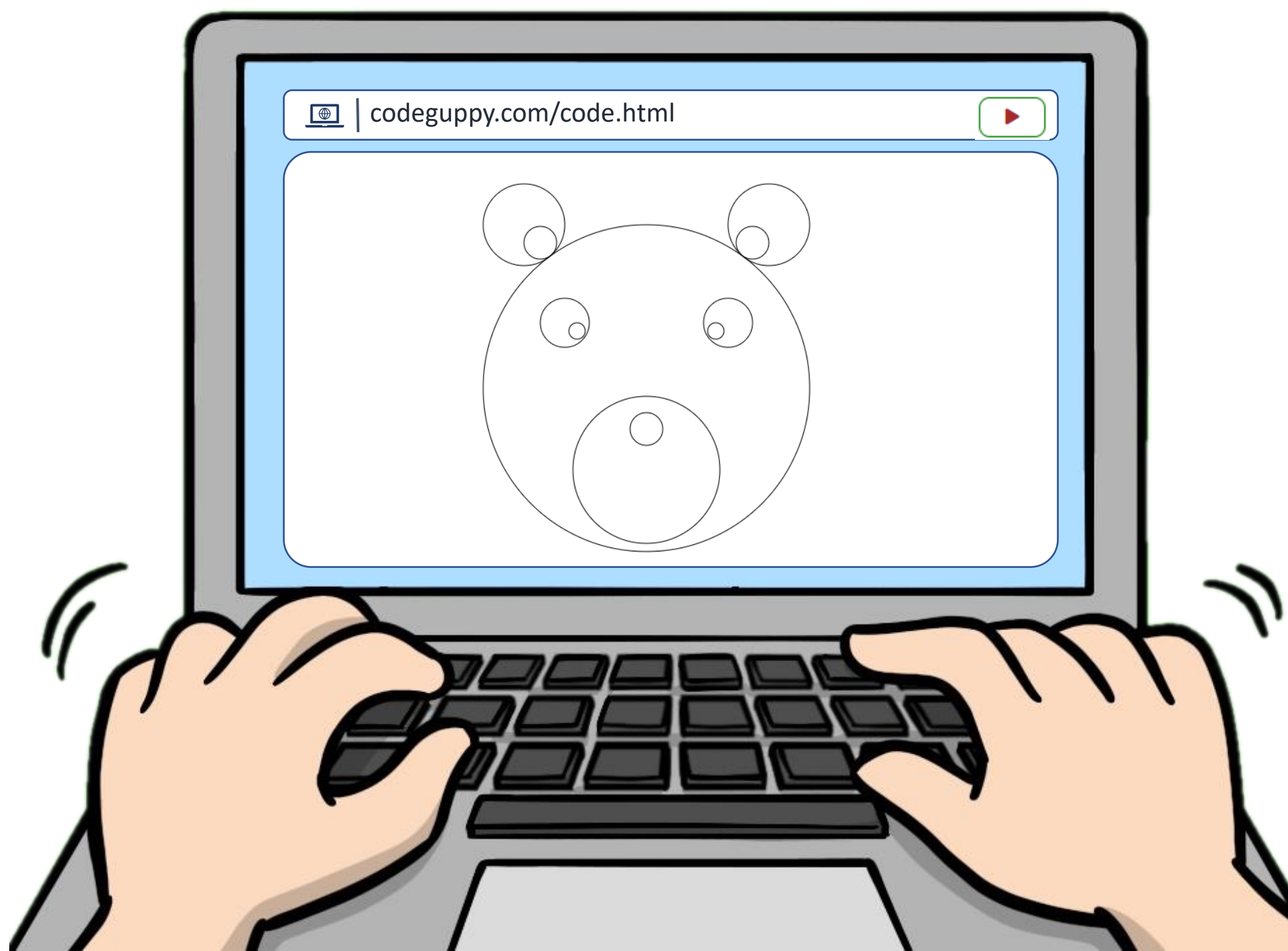
```
// Draw left ear  
circle(250, 100, 50);  
circle(270, 122, 20);
```

```
// Draw right ear  
circle(550, 100, 50);  
circle(530, 122, 20);
```

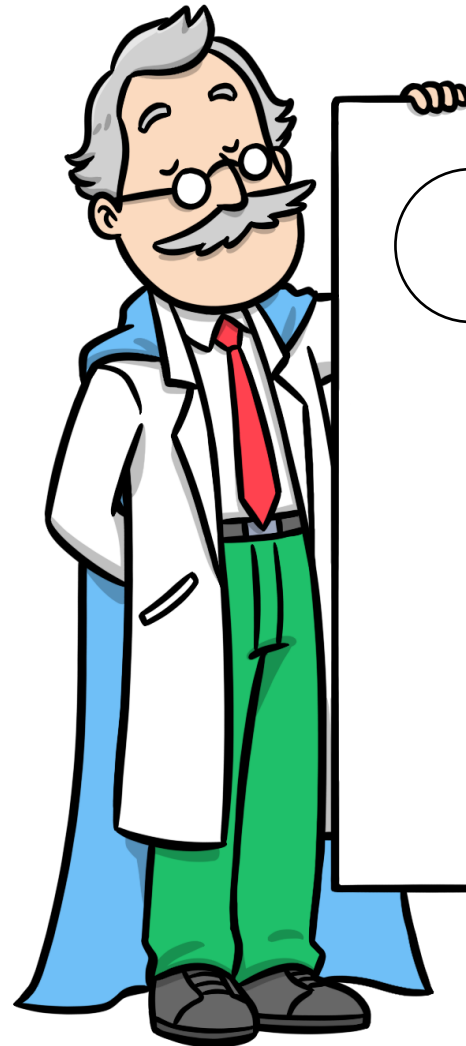
```
// Draw left eye  
circle(300, 220, 30);  
circle(315, 230, 10);
```

```
// Draw right eye  
circle(500, 220, 30);  
circle(485, 230, 10);
```

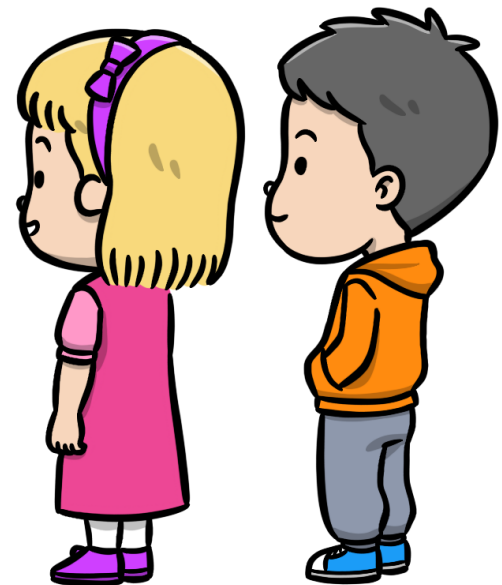
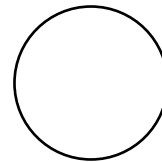
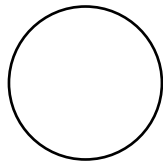
```
// Draw nose  
circle(400, 400, 90);  
circle(400, 350, 20);
```



# Homework



Write a program that  
makes a drawing using  
only circles.



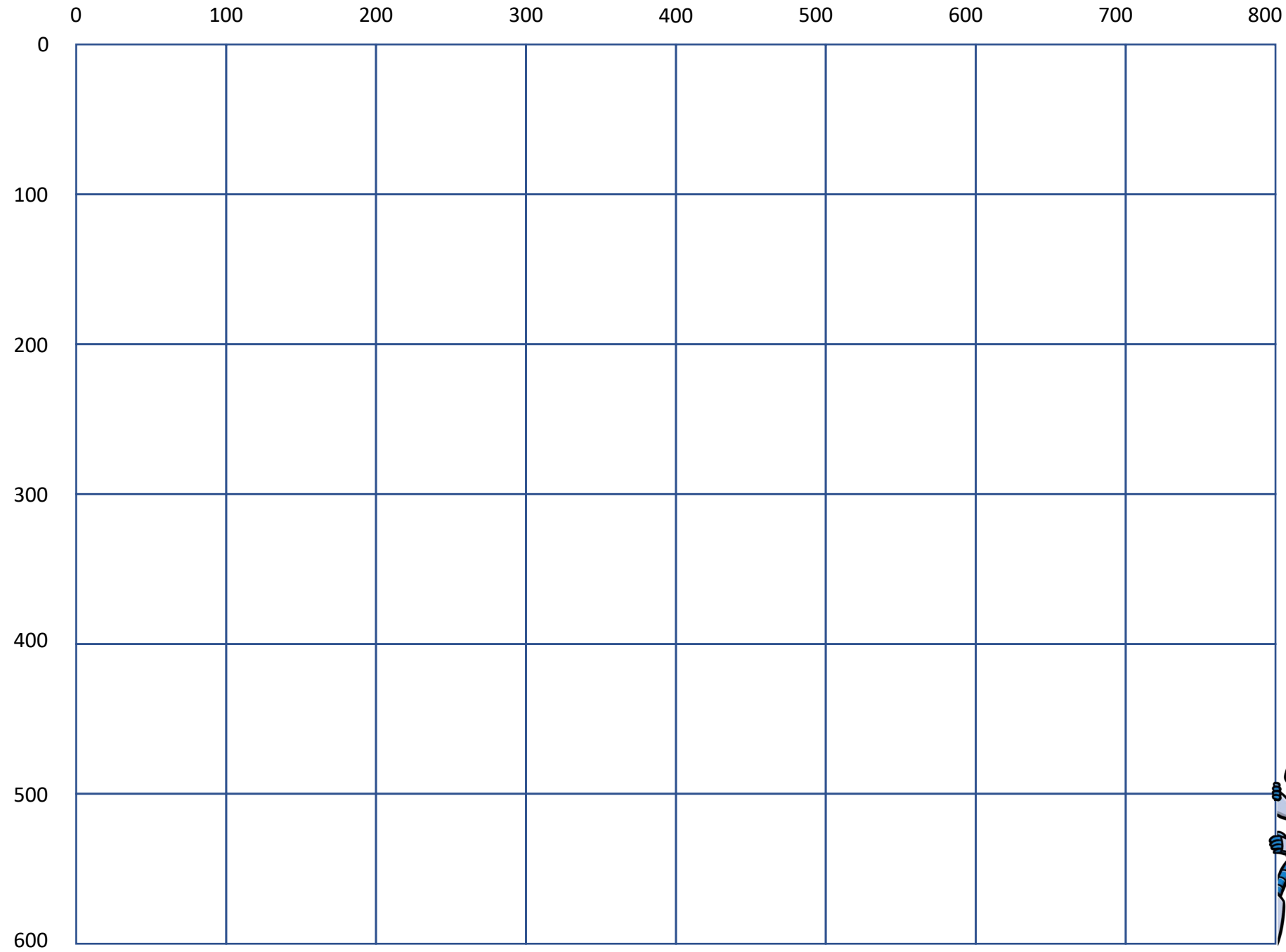


## Chapter III

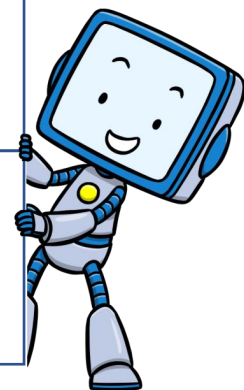
# Drawing shapes

- Remembering about canvas and circle instruction
- Learn how to draw other shapes  
(ellipse, rectangle, line, triangle, arc, point and text)
- A simple drawing with code program
- Homework





- In codeguppy.com, programs can write and draw on a graphical canvas of **800x600** pixels
- Origin is in the top-left corner
- Middle of the canvas is at about (400, 300)
- x coordinate goes from 0 to 800 (left to right)
- y coordinate goes from 0 to 600 (top to bottom)





Let's remember the "circle" instruction...

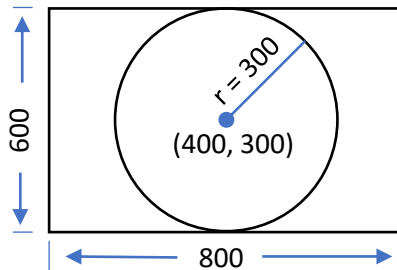
```
circle(400, 300, 300);
```

circle (x, y, r);

Parameters of the instruction.

There are 3 parameters inside **parenthesis** and separated by **comma**:

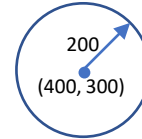
- 400, 300 - coordinates of circle center
- 300 – radius of the circle



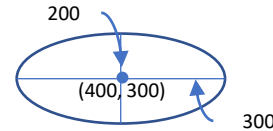
# Other graphical instructions



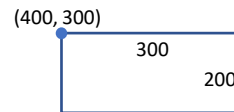
circle



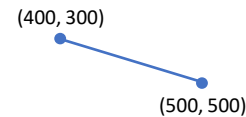
ellipse



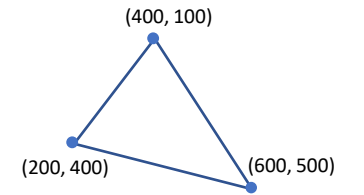
rect



line



triangle



arc

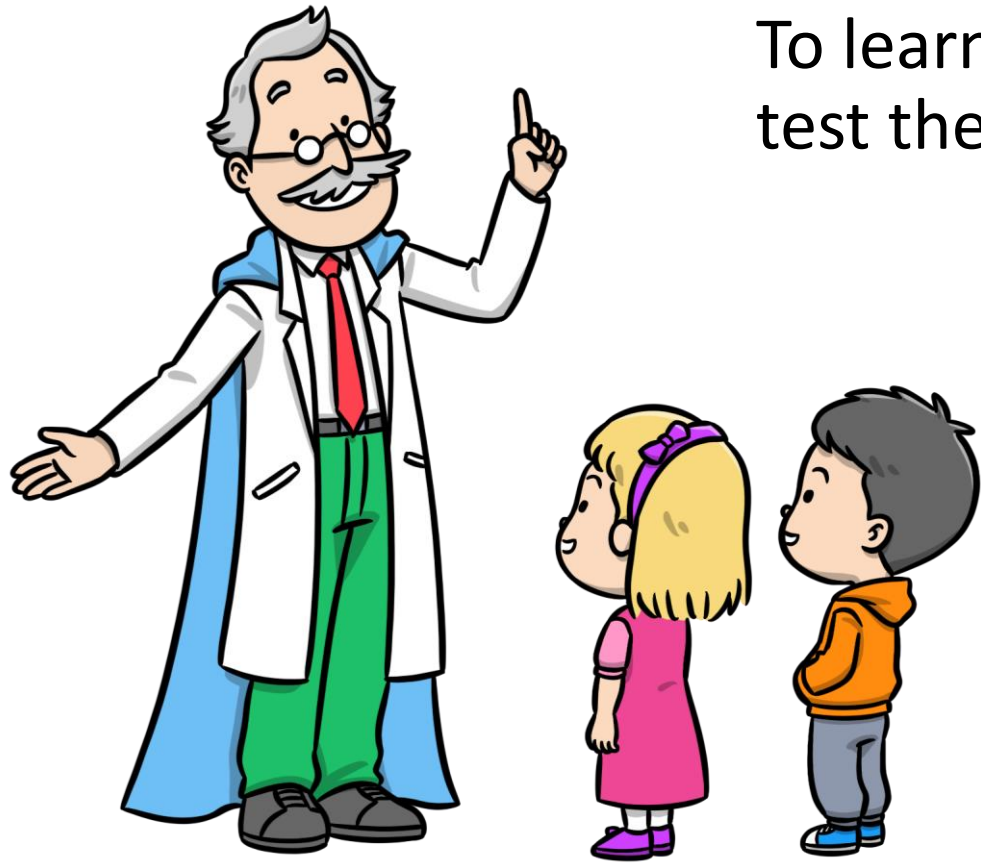


point



text

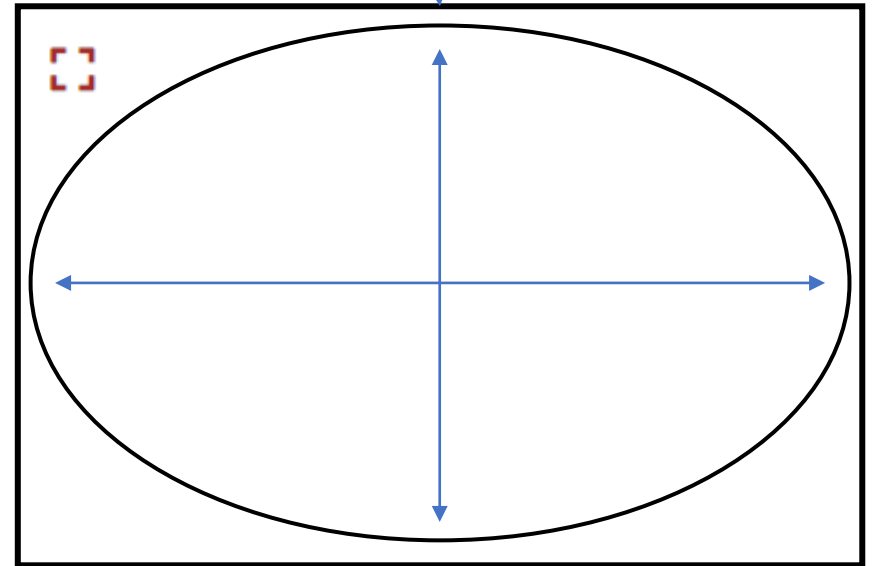
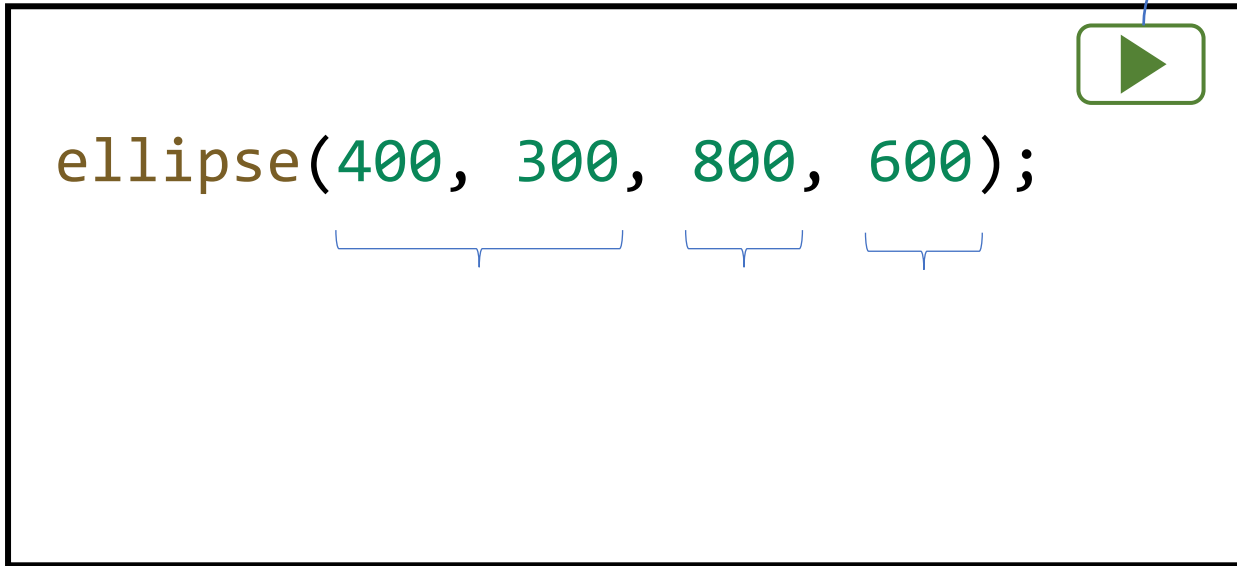




To learn the syntax of these instructions we will test them one by one in a new program.

# Let's draw an ellipse

```
ellipse(400, 300, 800, 600);
```



To draw an ellipse (aka an *elongated circle*), you use the instruction “ellipse” with 4 parameters:


- First 2 parameters: coordinates of the ellipse
- Third parameter: width of the ellipse
- Fourth parameter: height of the ellipse

This ellipse is big as the entire canvas!  
This is because:

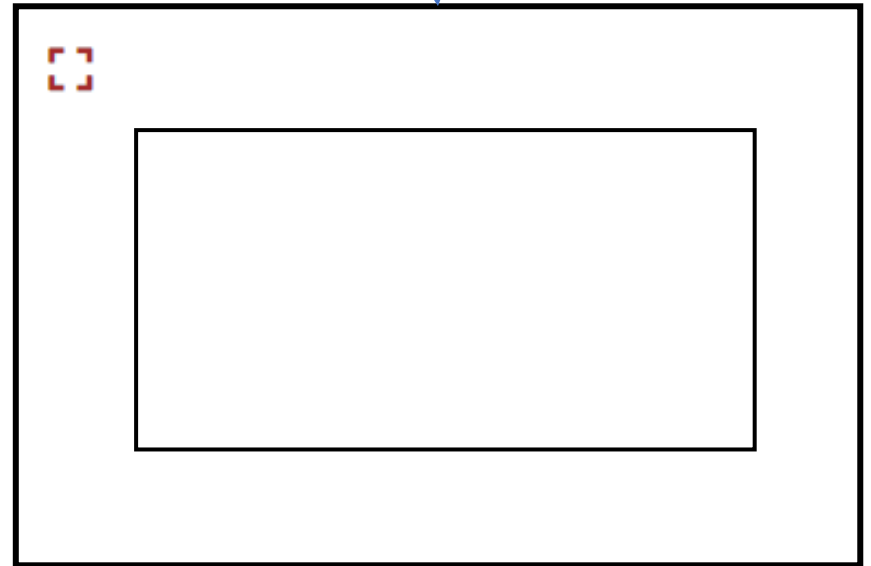
ellipse width = 800 (same as canvas width)  
ellipse height = 600 (same as canvas height)

# Let's draw a rectangle

```
rect(100, 100, 600, 400);
```



The diagram shows the parameters of the `rect` function: `100, 100, 600, 400`. Brackets are drawn under the first two values, with the label "x, y top-left corner" centered below them. Brackets are drawn under the third and fourth values, with the label "width" centered below the third and "height" centered below the fourth.



To draw rectangle, you use the instruction “**rect**”:

- First 2 parameters: top-left corner coordinates
- Third parameter: width of the rectangle
- Fourth parameter: height of the rectangle

The rectangle from this example is nicely centered on the canvas.

Can you tell why?

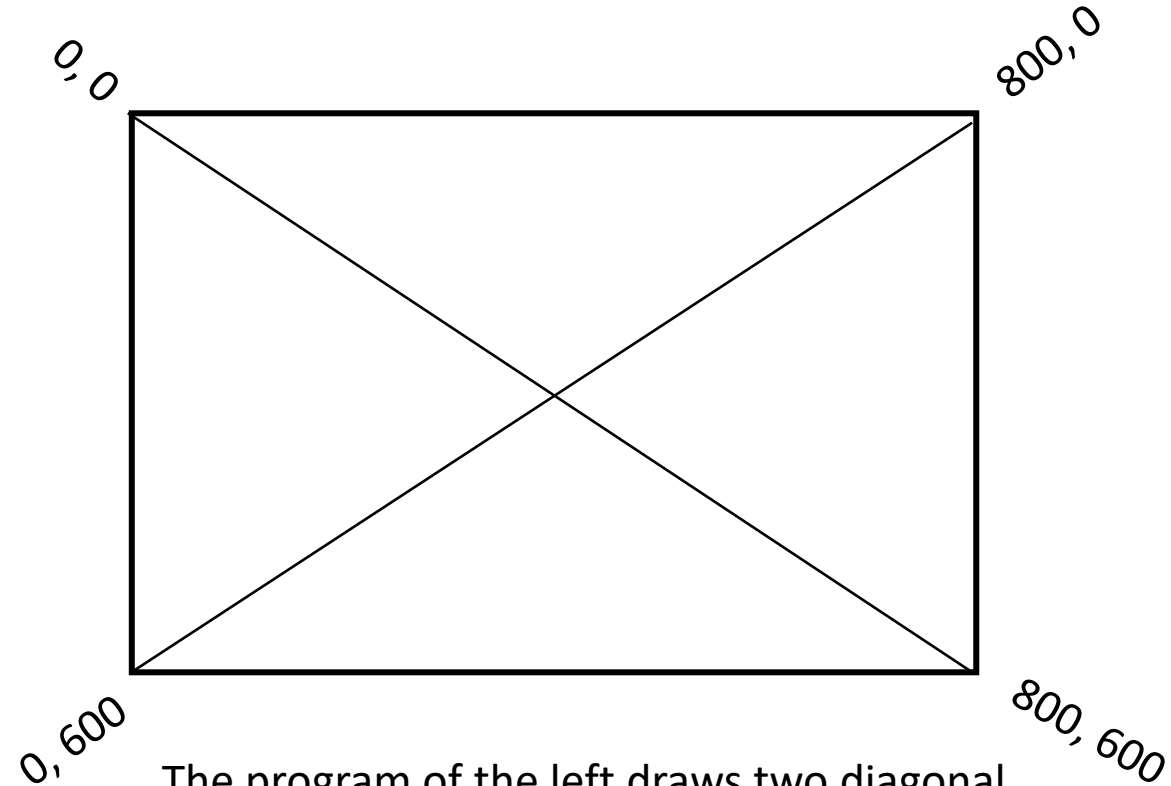
# Let's draw some lines

```
line(0, 0, 800, 600);  
line(0, 600, 800, 0);
```

x1, y1  
coordinates  
of line start

x2, y2  
coordinates  
of line end

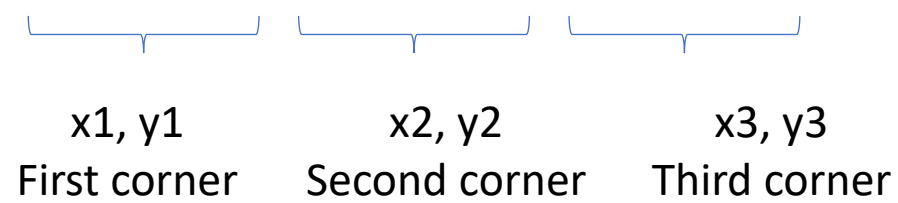
To draw rectangle, you use the instruction “**line**” and specify the coordinates x1, y1 and x2, y2 of the line points.



The program of the left draws two diagonal lines. Watch carefully and see that two opposite corners of the canvas are used as arguments in each instruction.

# Let's draw a triangle

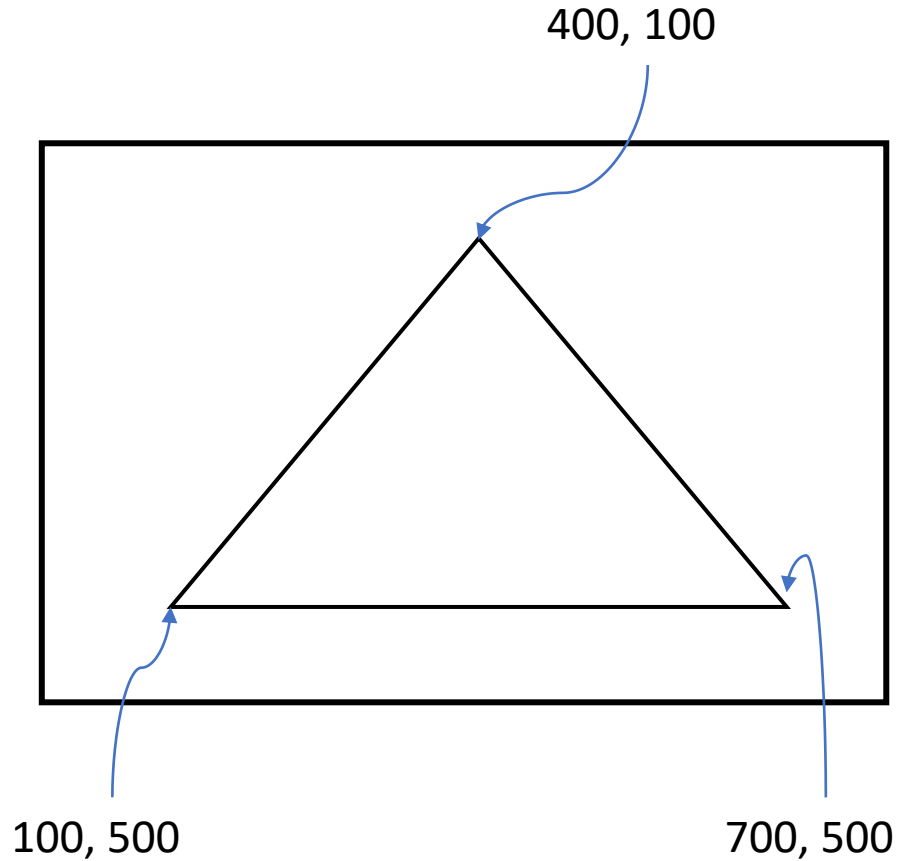

```
triangle(400, 100, 100, 500, 700, 500);
```



x1, y1  
First corner

x2, y2  
Second corner

x3, y3  
Third corner



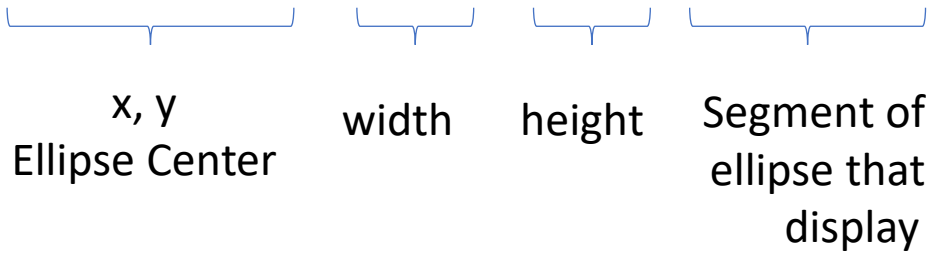
Triangle is an instruction that takes lots of parameters!

But they are very simple: they are the x, y coordinates of the 3 corners of the triangle. In total 6 numbers!



# Let's draw an arc

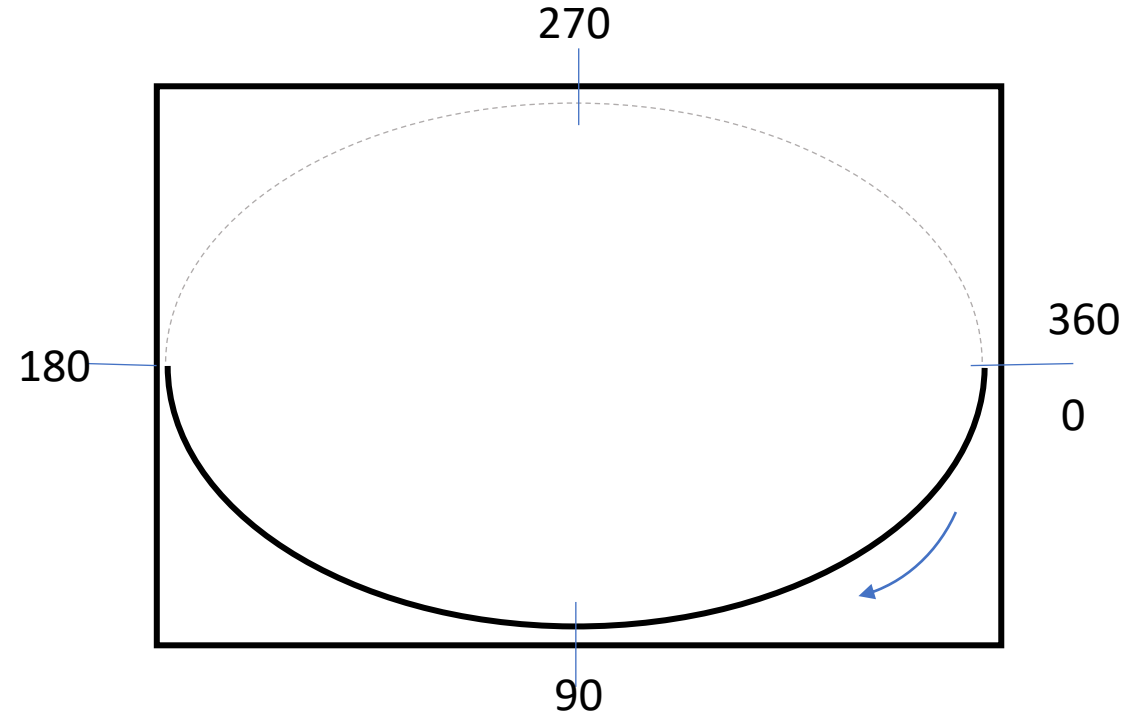
```
arc(400, 300, 800, 600, 0, 180);
```



The diagram shows the parameters of the `arc` function grouped with blue brackets and labeled below:

- `400, 300`: x, y Ellipse Center
- `800, 600`: width height
- `0, 180`: Segment of the ellipse that we display

A green play button icon is located in the top right corner of the code block.



To draw an arc, you need to imagine an ellipse!

The first 4 parameters of “arc” instructions are defining the virtual ellipse. The ellipse is just imaginary.

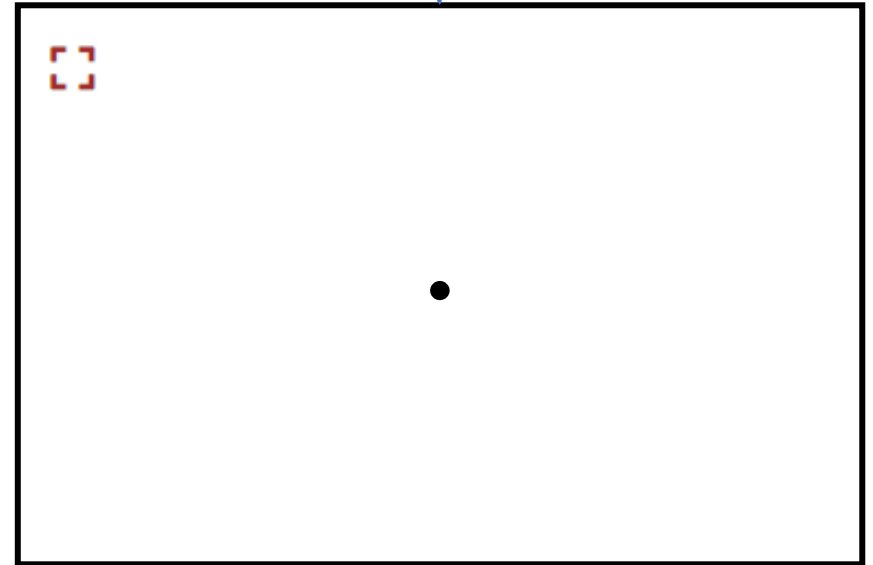

Then the last two parameters are specifying what *segment* of the ellipse to be displayed. Here the numbers are from 0 to 360 -- and are trigonometric degrees!

If you run the program, you'll see an arc that looks like the bottom half of the ellipse (see numbers 0, 180)

# Let's draw a single point!

```
point(400, 300);
```

x, y  
coordinates



Sometimes, you need to draw a single tiny point.  
You can do this using the “point” instruction.

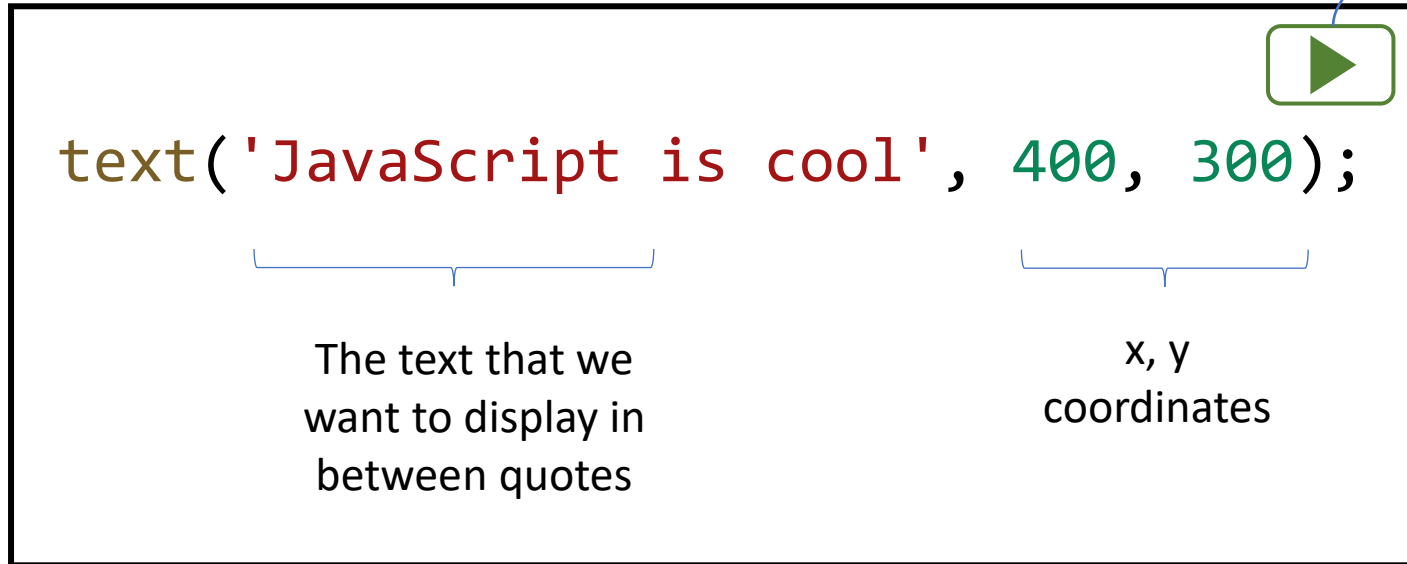
It only takes two parameters: the coordinates of the point.

Watch carefully in the middle of the canvas!

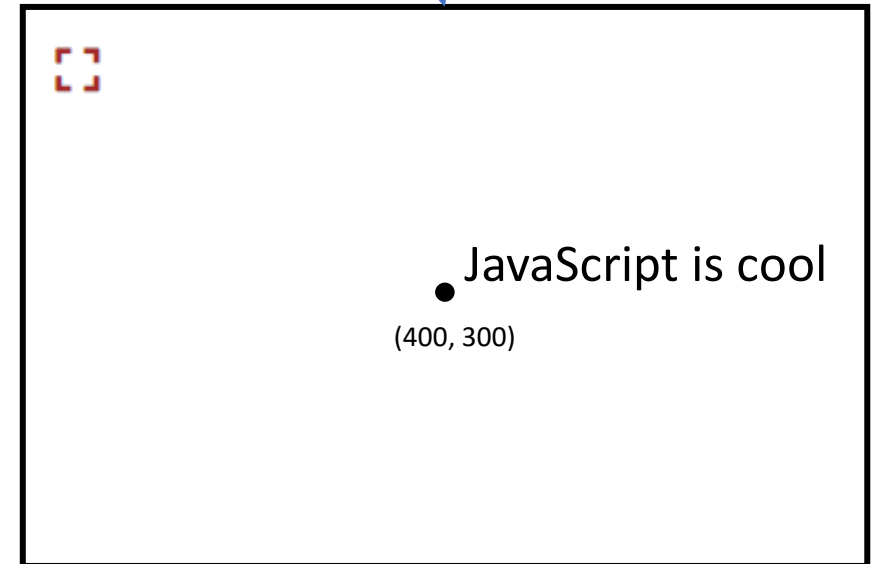
You should see the point, although is very tiny!

# How about adding some text?

```
text('JavaScript is cool', 400, 300);
```



The diagram shows the function call `text('JavaScript is cool', 400, 300);`. A blue bracket under the string `'JavaScript is cool'` is labeled "The text that we want to display in between quotes". Another blue bracket under the numbers `400, 300` is labeled "x, y coordinates". A green play button icon is in the top right corner of the box.



The diagram shows a canvas with a red dashed box in the top left corner. The text "JavaScript is cool" is displayed on the canvas. A black dot is placed at the start of the text, with the coordinates "(400, 300)" written below it. A blue arrow points from the play button in the previous diagram to this canvas.

To display the text, you need to specify the text, in between quotes (you can use single or double quotes) as well as the coordinates where you want to display the text.

The text is displayed at specified coordinates!

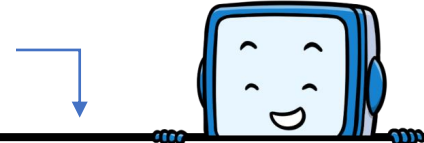
Note: Don't put any other quotes or other funny symbols inside the text! Always close the text with the same quote you started.

It is necessary to enclose the text you want to display in between quotes.

Use either single or double quote, but don't mix them in the same message.



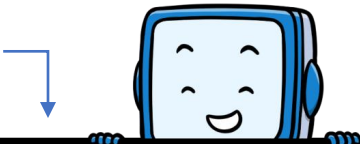
Use double quotes



```
text("JavaScript", 400, 300);
```

OR

Use single quotes



```
text('JavaScript', 400, 300);
```

# Did you notice the pattern of these JavaScript instructions?



## If instruction has no parameters

You type the name of instruction followed by ();

instruction ( ) ;

## If instruction has parameters

You put parameters inside () separated by ,

instruction ( "Hello" , 200 ) ;

circle text

ellipse point

rect line arc  
triangle

# Graphical instructions reference



```
circle(400, 300, 200);
```

```
ellipse(400, 300, 300, 200);
```

```
rect(400, 300, 300, 200);
```

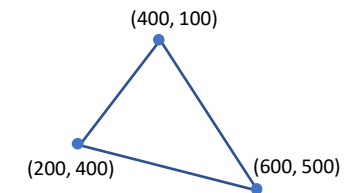
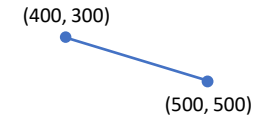
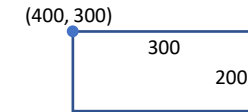
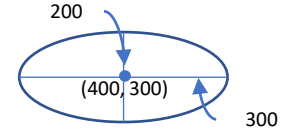
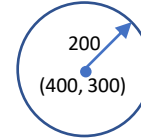
```
line(400, 300, 500, 500);
```

```
triangle(400, 100, 200, 400, 600, 500);
```

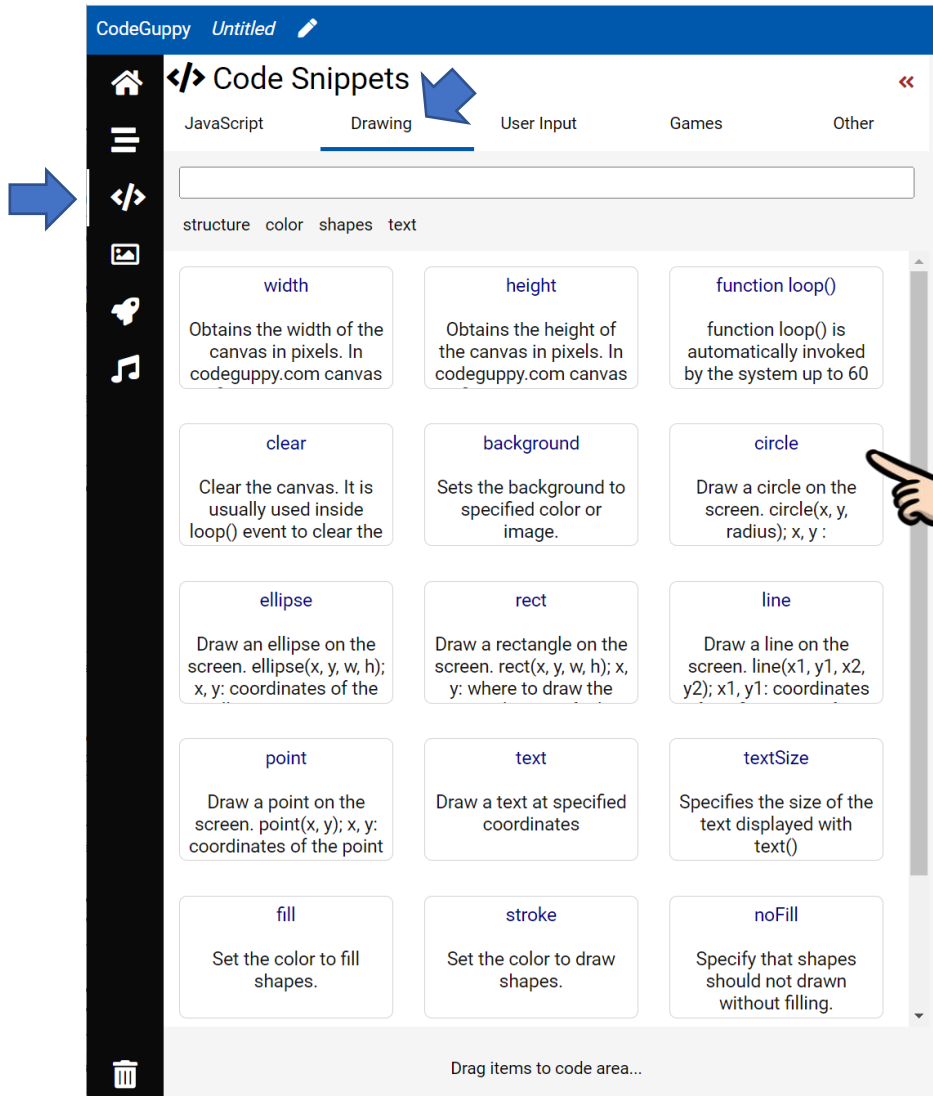
```
arc(400, 300, 300, 200, 0, 180);
```

```
point(400, 300);
```

```
text('JavaScript', 400, 300);
```



# What if I forget the syntax of these instructions?



If you can forget the syntax of these graphical instructions, just open the “Code Snippets” palette and go to the “Drawing” tab.

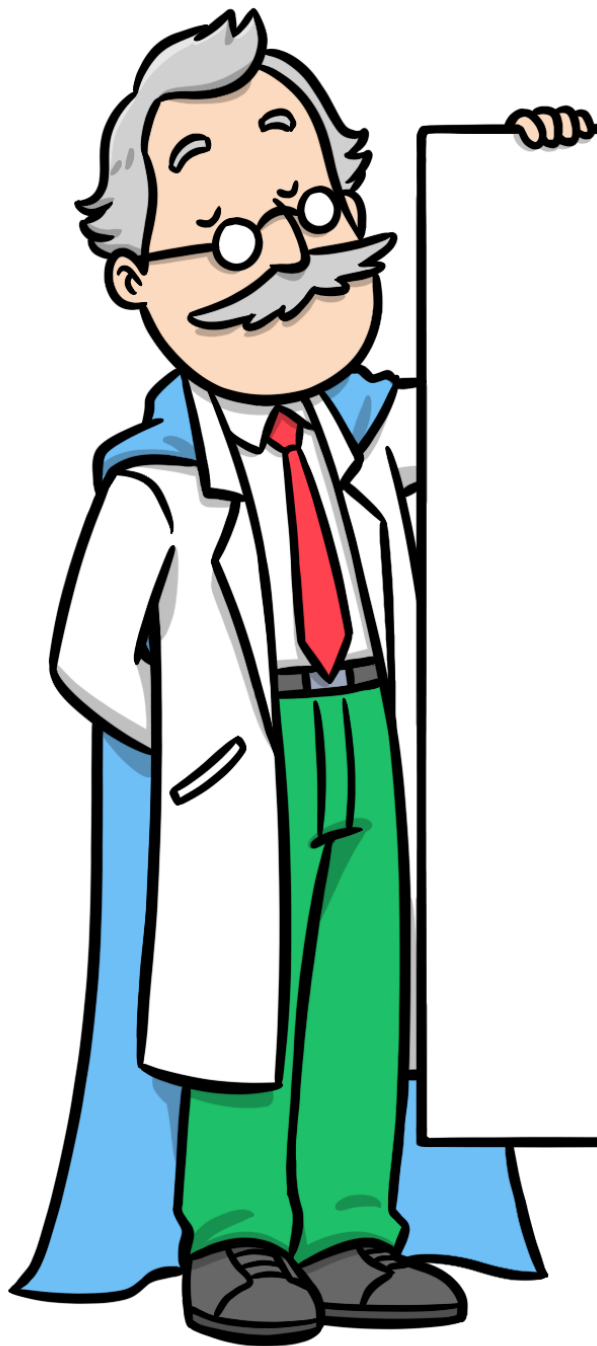
You’ll be able to drag&drop in your code, small snippets of code as needed.

Note: This palette contains also other commands that we’ll learn about in the future.



Let's draw with code by combining all these new instructions in a bigger program...





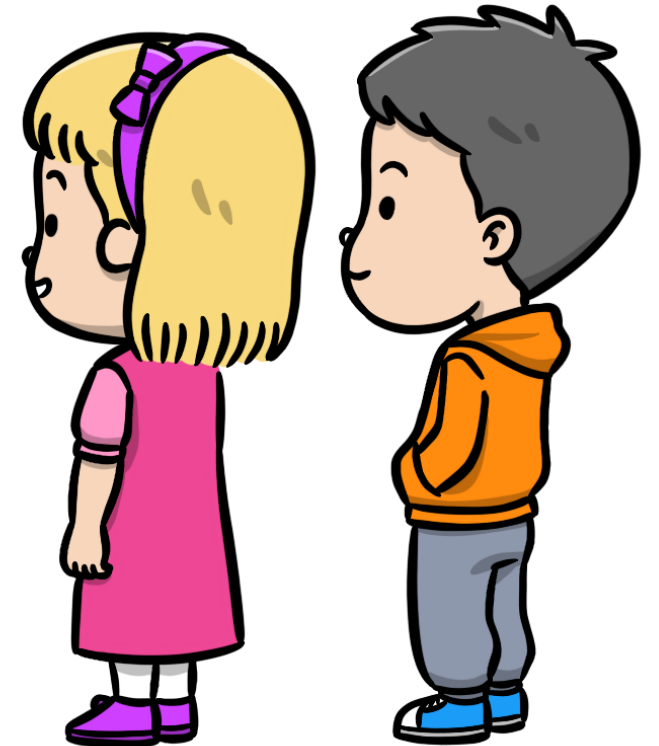
```
// Street
line(0, 500, 800, 500);

// Sun
circle(750, 50, 150);
line(480, 60, 561, 47);
line(548, 224, 602, 172);
line(740, 304, 747, 236);

// Car
rect(175, 340, 223, 54);
rect(108, 394, 362, 74);
circle(168, 468, 32);
circle(408, 468, 32);
```

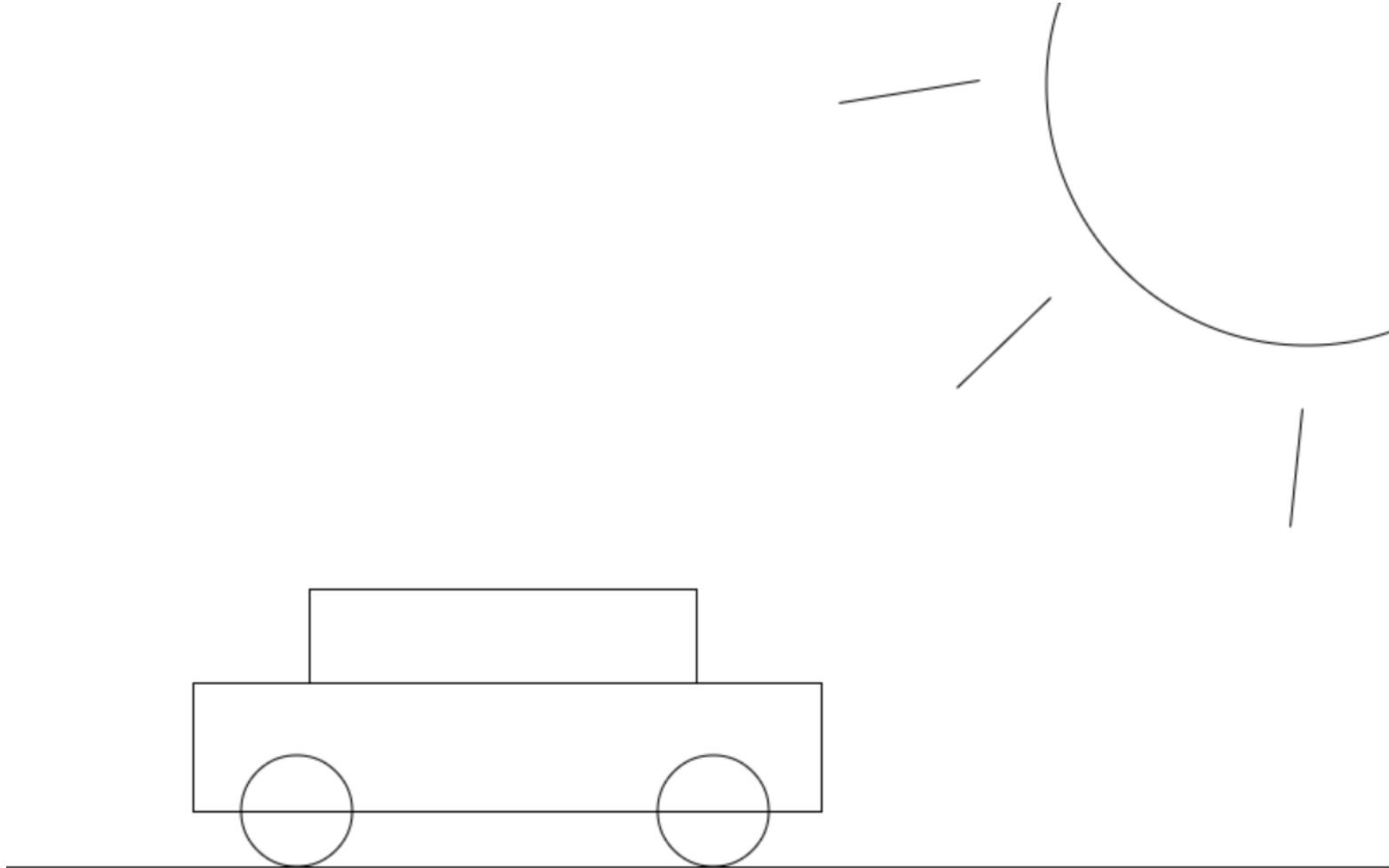


Drawing with code  
(a type-in program)



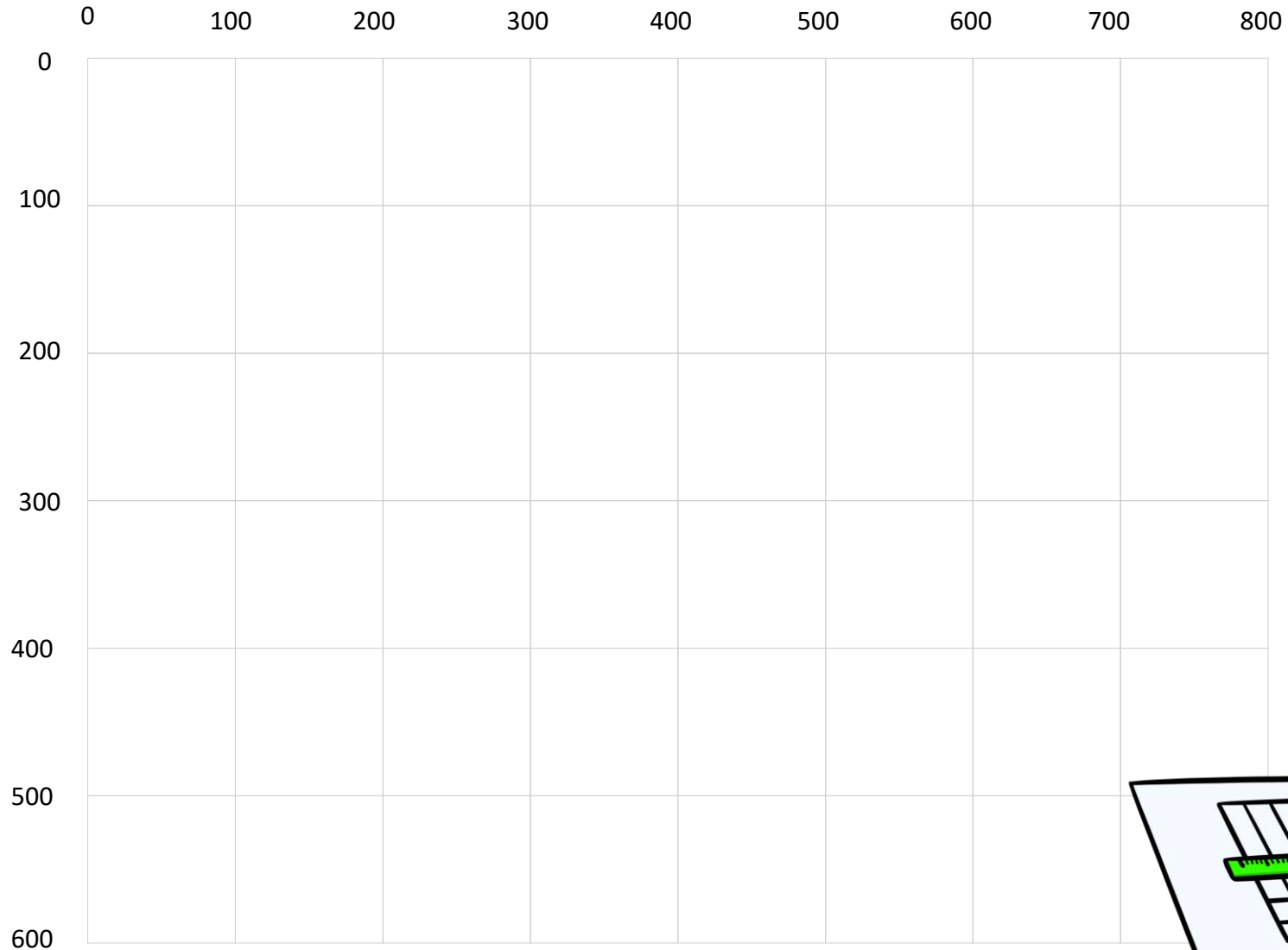


```
// Street  
line(0, 500, 800, 500);  
  
// Sun  
circle(750, 50, 150);  
line(480, 60, 561, 47);  
line(548, 224, 602, 172);  
line(740, 304, 747, 236);  
  
// Car  
rect(175, 340, 223, 54);  
rect(108, 394, 362, 74);  
circle(168, 468, 32);  
circle(408, 468, 32);
```



If you typed in the program correctly you should see a car a nice drawing with a car and a sun!  
For now, the drawing is black and white. Later in this course, we will learn on how to add color to shapes.

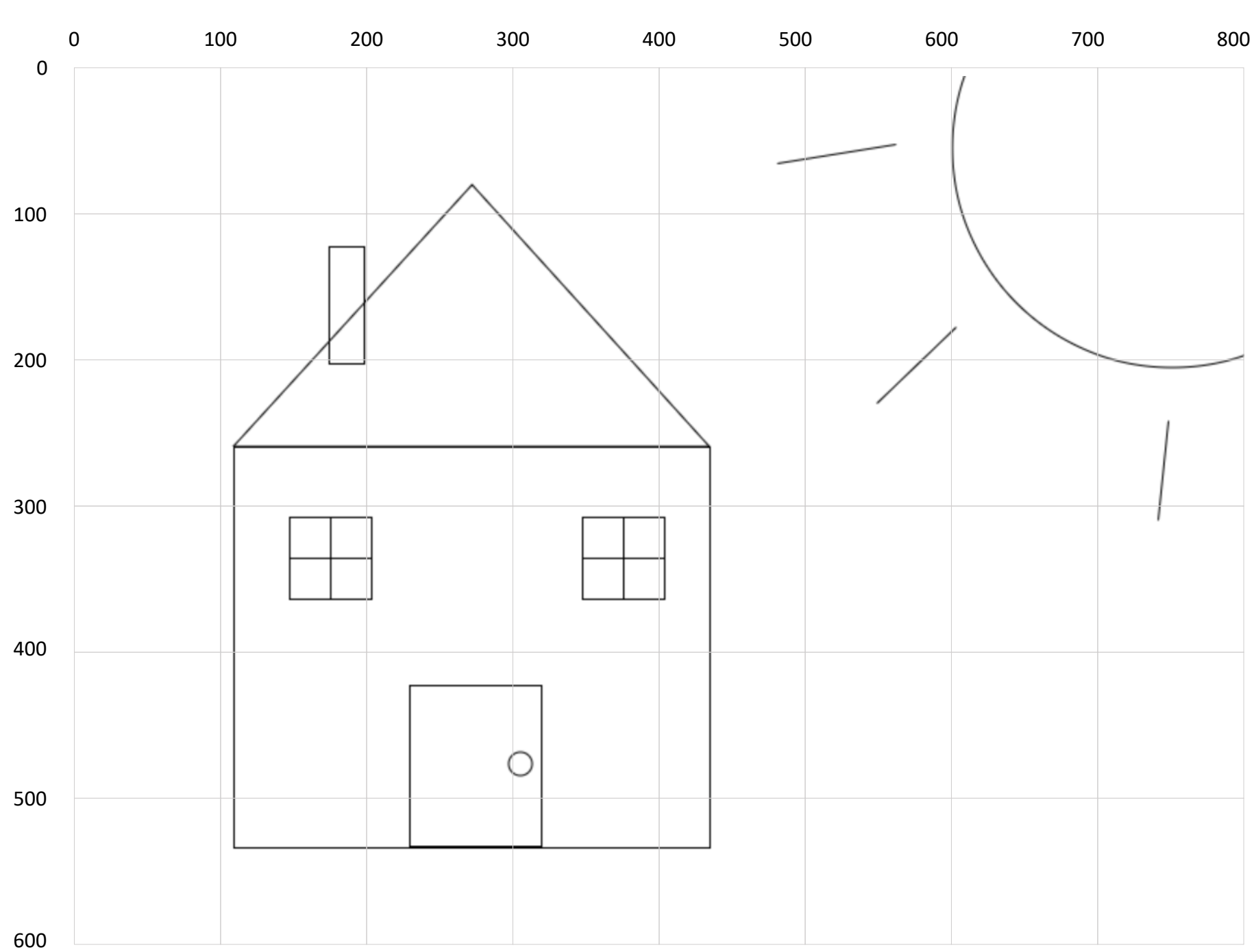
# Homework



On a blank piece of paper draw simulate our 800x600 pixels canvas by drawing an 8 x 6 inches rectangle.

Draw thin dividing lines every 1 inch ...

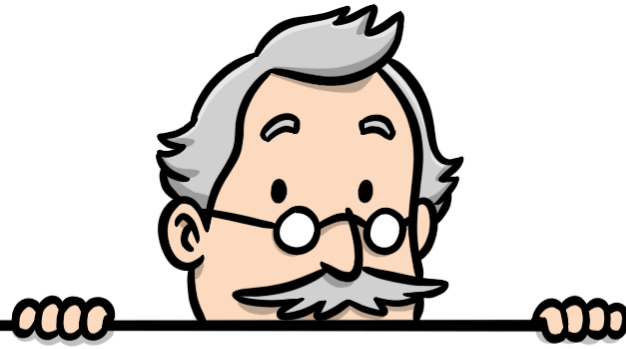




# Homework

Using only basic shapes, draw a scene (e.g. a house, a flower, a robot, etc.)

Then write a JavaScript program that draws with code your scene!



## Chapter IV

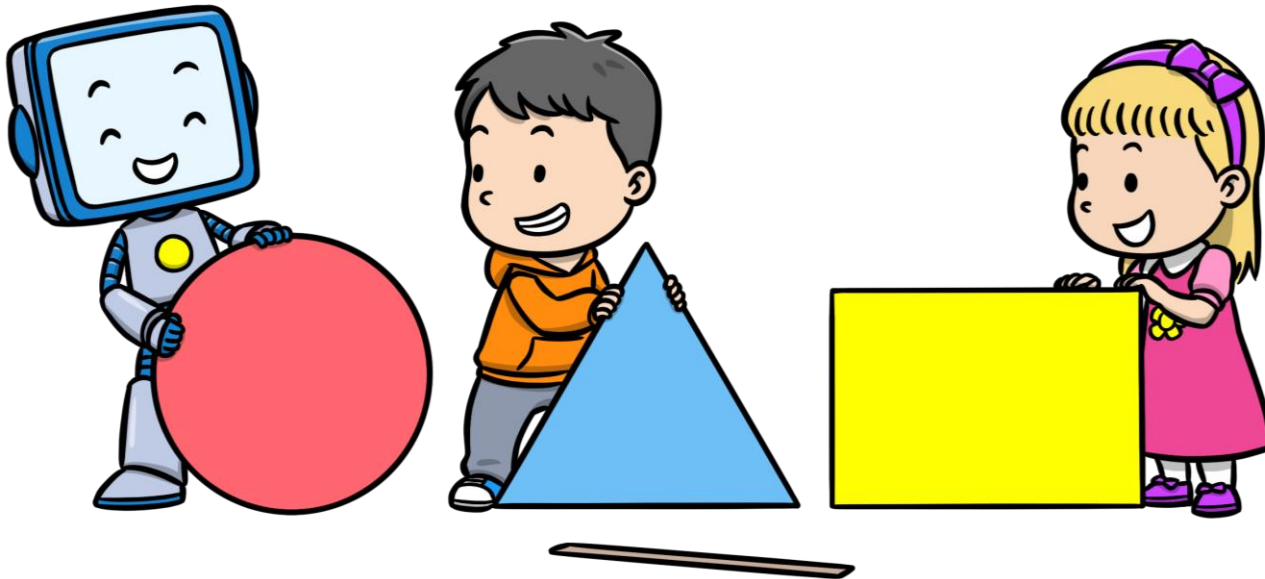
### Shape attributes

- About colors
- Drawing colored shapes
- Setting line thickness
- Text attributes
- Drawing complex scenes with code
- Homework

# What have we learned about?

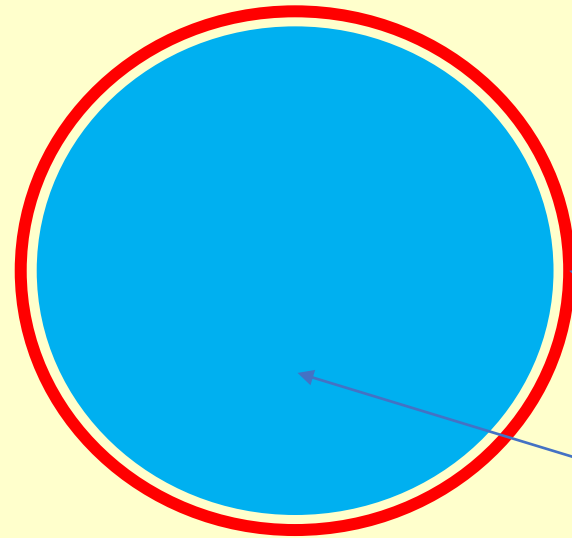
- Until now we learned about eight shape drawing instructions and how to use them to draw black and white shapes on our 800x600 pixels canvas.

... let's see now how to add some color to our drawings!

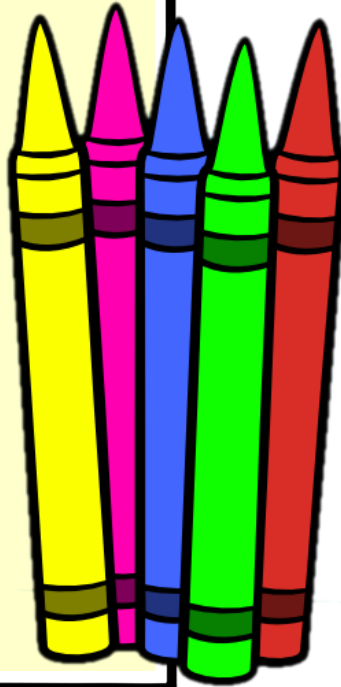


```
circle(400, 300, 200);  
ellipse(400, 300, 300, 200);  
rect(400, 300, 300, 200);  
line(400, 300, 500, 500);  
triangle(400, 100, 200, 400, 600, 500);  
arc(400, 300, 300, 200, 0, 180);  
point(400, 300);  
text('JavaScript', 400, 300);
```

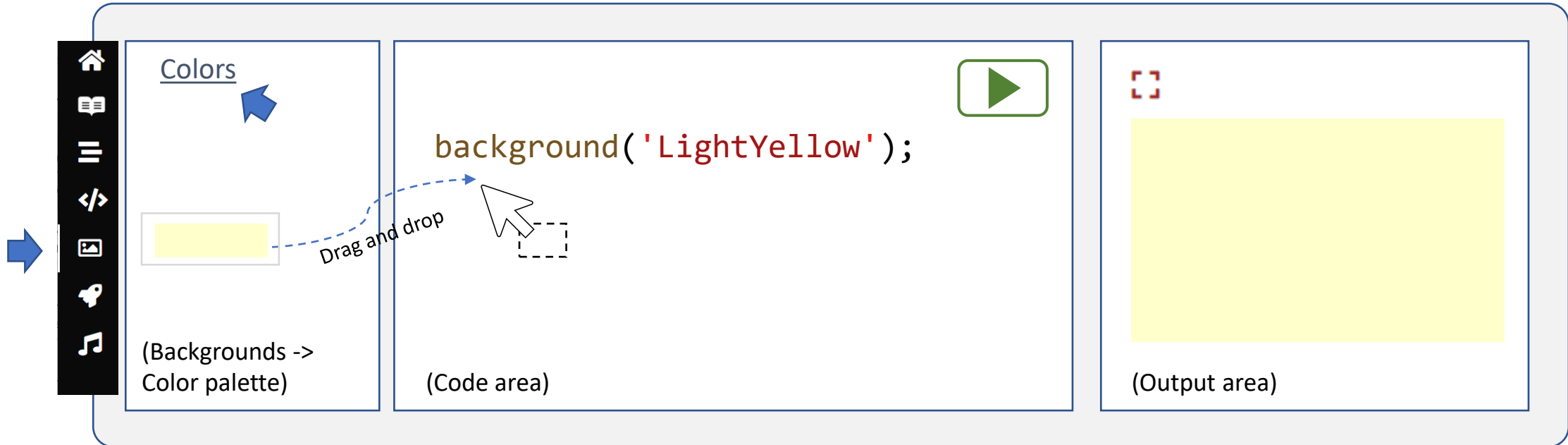
Let's see how to add some color...




- Set background / canvas color
- Set shape outline color
- Set shape fill color



# First, let's set the background color...

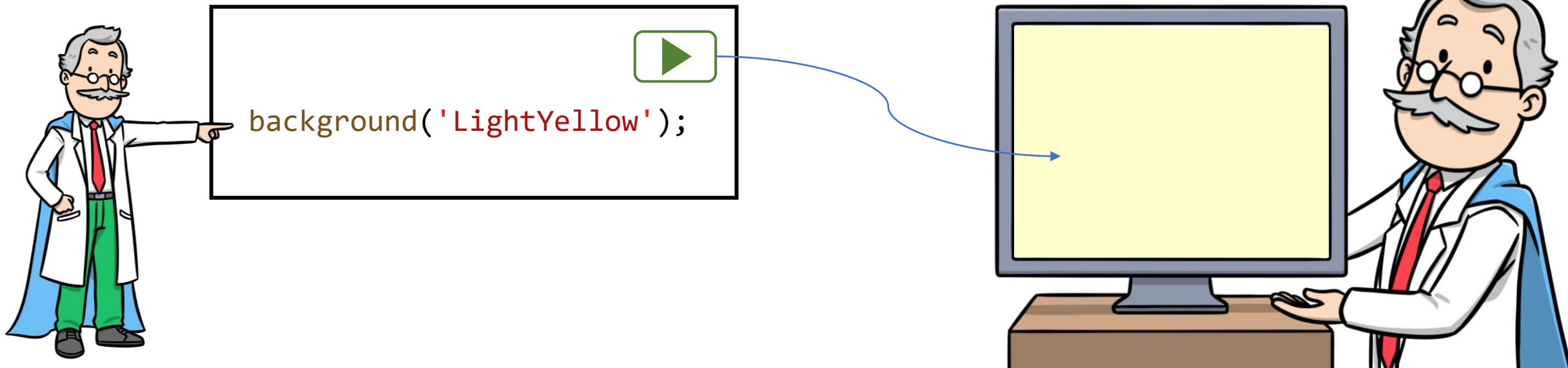


- In the code editor, open the Background Palette
- Go to the “Colors” tab
- Select a nice color and drag it into the code area
- Notice the code it creates there...
- Now Run the program 
- You should see the canvas changing color in accordance with the selected color
- Press “Stop” to stop the program and clear the canvas...



# What is background() doing?

- background() is a special instruction that sets the color of the canvas
- background() takes as parameter a color in between single or double quotes: `background("lightyellow");`
- By default, if no background() instruction is used, the canvas will appear white
- Note: You probably noticed on the backgrounds palette that you can also use an image as a background. Feel free to explore this feature on your own. For now, we are interested only in setting the background to a solid color.

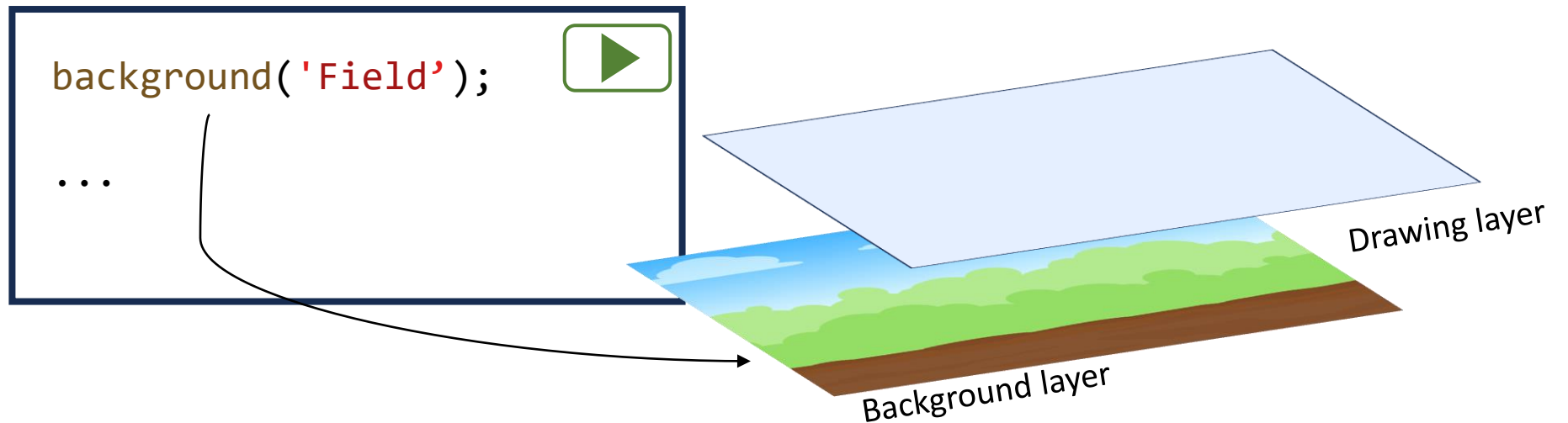


# Multiple drawing layers

The codeguppy.com system has a multi-layered drawing architecture. For instance, the **background** command is impacting the bottom layer, while all the other shape drawing commands are operating on the top drawing layer.

In this way if you change the background after you have something drawn, the command won't interfere with your drawing.

Sprites are also operating in a different layer... but we'll learn more about sprites in a future lesson.




# Now, let's set the outline color (aka the stroke)...

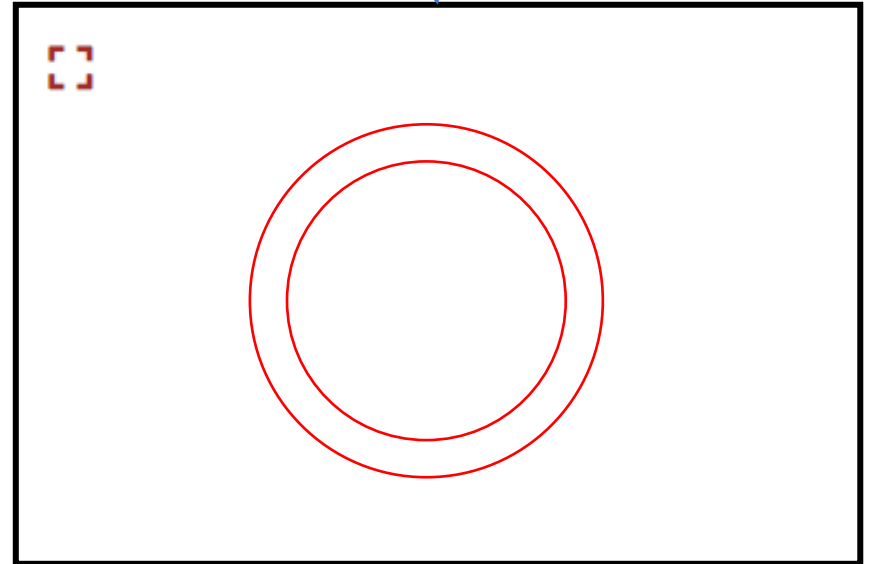
```
stroke("red");
```

```
circle(400, 300, 200);
```

```
circle(400, 300, 180);
```



- This program has 3 lines
- Please type it carefully as you see on the screen
- When ready, press "Run" to execute the program 



- You should see 2 concentric circles, both drawn in red.
- Why both circles are red?

# What is stroke() doing?

- stroke() is a special instruction that sets the color of the shape outline (aka stroke).
- stroke() takes as parameter a color in between quotes: `stroke("red");`
- Once a color is selected, it is persisted and used to draw all shapes on the screen, until a new color is selected with other stroke() instruction
- By default, if no stroke() instruction is used, the program draw black shapes (as saw in previous lesson)



```
stroke("red");  
circle(400, 300, 200);  
circle(400, 300, 180);  
  
stroke("blue");  
rect(50, 50, 700, 500);
```



← Type-in this program to see the effect...

# Important: Numbers and strings



```
circle(400, 300, 200);
```



```
stroke("Red");
```



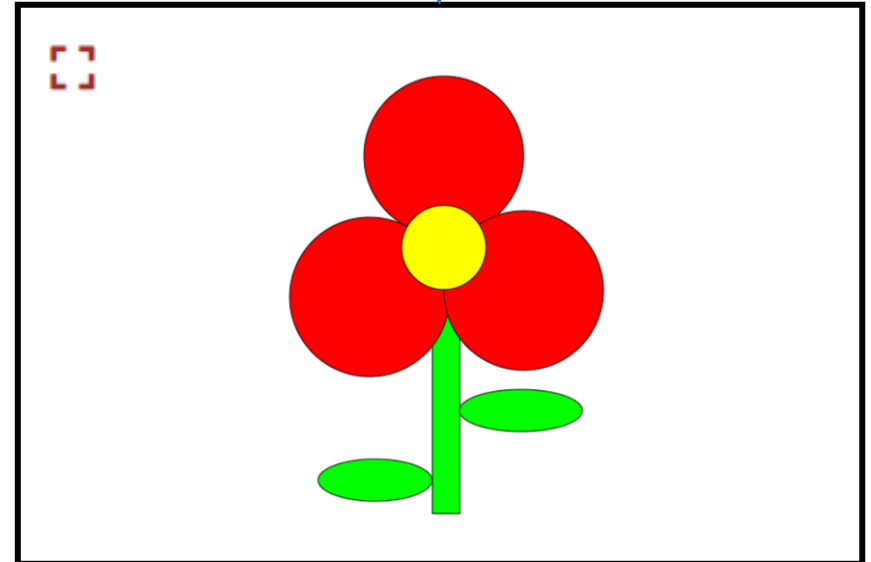
Until now we mostly worked with instructions that took numbers as parameters (e.g. *circle*).

As you saw until now, in this lesson we will encounter a series of instructions that takes text (aka *strings*) as arguments.


Strings are small text snippets enclosed by single or double quotes. When you see them in a program, please type them as is and don't forget the quotes.

# Next, let's set the fill color...

```
// Stem  
fill("lime");  
rect(277, 313, 30, 237);  
ellipse(215, 514, 124, 46);  
ellipse(374, 438, 134, 46 );  
  
// Flower  
fill("red");  
circle(290, 160, 87);  
circle(209, 314, 87);  
circle(377, 307, 87);  
fill("yellow");  
circle(290, 260, 46);
```

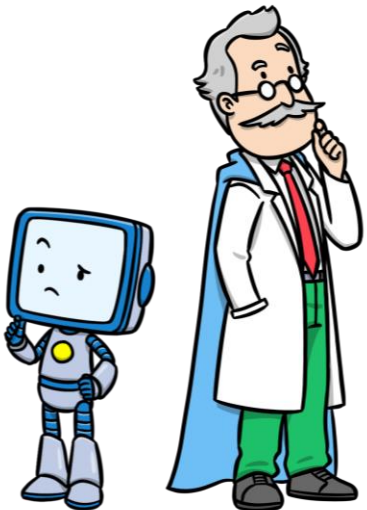


- This is a nice drawing!

- This is a longer program, but the effect is worth the typing. Press "Run" when ready... 

# What is fill() doing?

- fill() is an instruction like stroke(), but instead of setting the color for the outline, it sets the color for the interior
- fill() has the same syntax as stroke() taking as parameter a color in between quotes: `fill("red");`
- Once a color is selected, it is persisted and used to fill the interior of all new shapes, until a new color is selected with other fill() instruction
- By default, if no fill() instruction is used, the program draws empty shapes (with transparent interior)



You may ask: what other colors can we specify as parameter to background(), stroke() and fill() instructions?

I'm glad you asked...

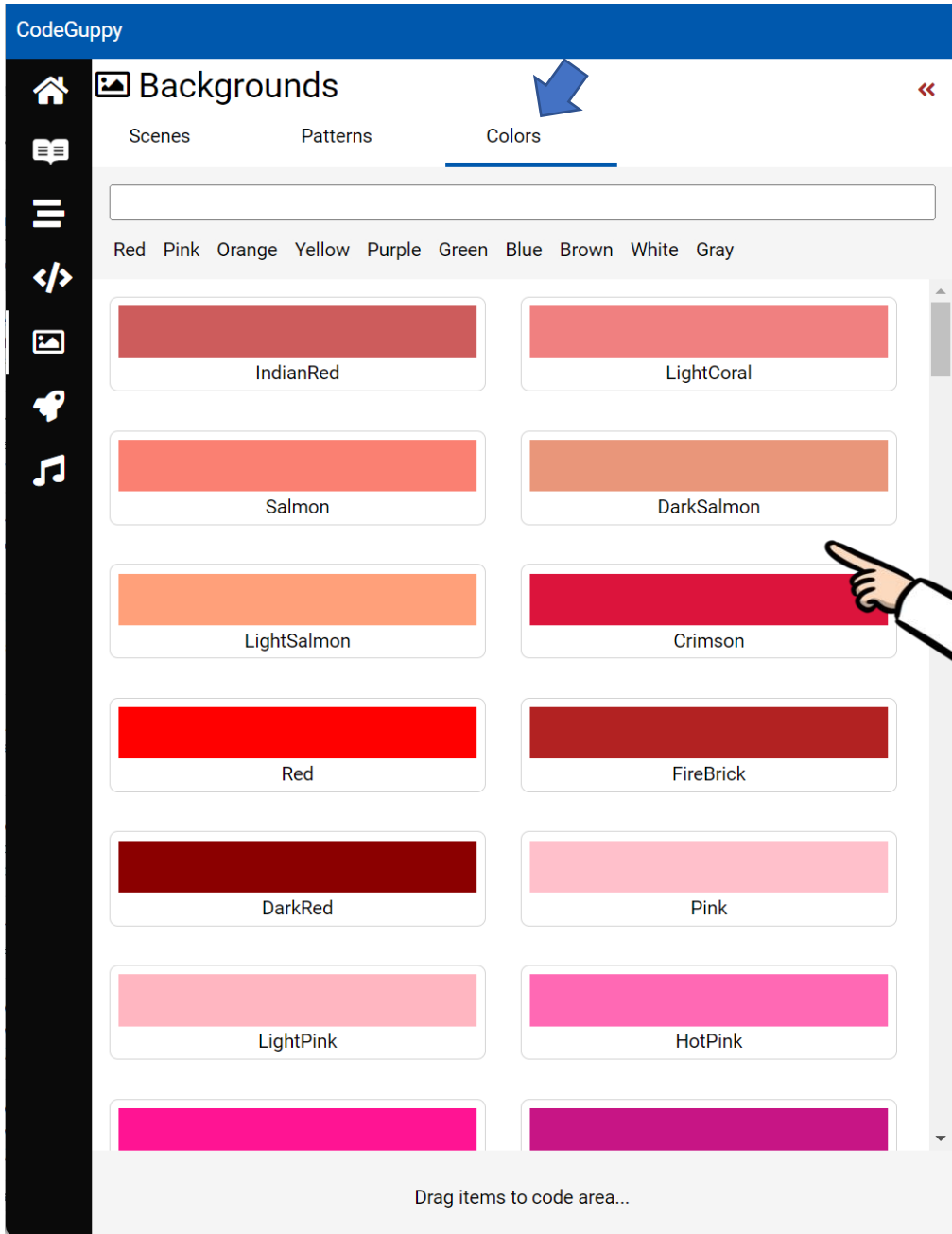
# There are plenty of colors and shades you can choose...



- IndianRed, LightCoral, Salmon, DarkSalmon, LightSalmon, Crimson, Red, FireBrick, DarkRed
- Pink, LightPink, HotPink, DeepPink, MediumVioletRed, PaleVioletRed
- LightSalmon, Coral, Tomato, OrangeRed, DarkOrange, Orange
- Gold, Yellow, LightYellow, LemonChiffon, LightGoldenrodYellow, PapayaWhip, Moccasin, PeachPuff, PaleGoldenrod, Khaki, DarkKhaki
- Lavender, Thistle, Plum, Violet, Orchid, Fuchsia, Magenta, MediumOrchid, MediumPurple, RebeccaPurple, BlueViolet, DarkViolet, DarkOrchid, DarkMagenta, Purple, Indigo, SlateBlue, DarkSlateBlue, MediumSlateBlue
- GreenYellow, Chartreuse, LawnGreen, Lime, LimeGreen, PaleGreen, LightGreen, MediumSpringGreen, SpringGreen, MediumSeaGreen, SeaGreen, ForestGreen, Green, DarkGreen, YellowGreen, OliveDrab, Olive, DarkOliveGreen, MediumAquamarine, DarkSeaGreen, LightSeaGreen, DarkCyan, Teal
- Aqua, Cyan, LightCyan, PaleTurquoise, Aquamarine, Turquoise, MediumTurquoise, DarkTurquoise, CadetBlue, SteelBlue, LightSteelBlue, PowderBlue, LightBlue, SkyBlue, LightSkyBlue, DeepSkyBlue, DodgerBlue, CornflowerBlue, MediumSlateBlue, RoyalBlue, Blue, MediumBlue, DarkBlue, Navy, MidnightBlue
- Cornsilk, BlanchedAlmond, Bisque, NavajoWhite, Wheat, BurlyWood, Tan, RosyBrown, SandyBrown, Goldenrod, DarkGoldenrod, Peru, Chocolate, SaddleBrown, Sienna, Brown, Maroon
- White, Snow, HoneyDew, MintCream, Azure, AliceBlue, GhostWhite, WhiteSmoke, SeaShell, Beige, OldLace, FloralWhite, Ivory, AntiqueWhite, Linen, LavenderBlush, MistyRose
- Gainsboro, LightGray, Silver, DarkGray, Gray, DimGray, LightSlateGray, SlateGray, DarkSlateGray, Black

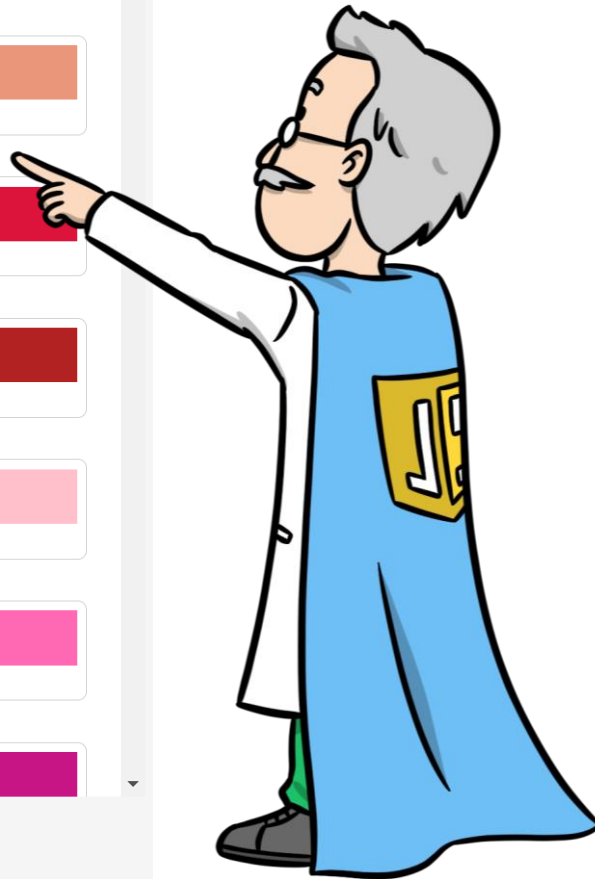
Any color on this slide, can be used as parameter to stroke() and fill()



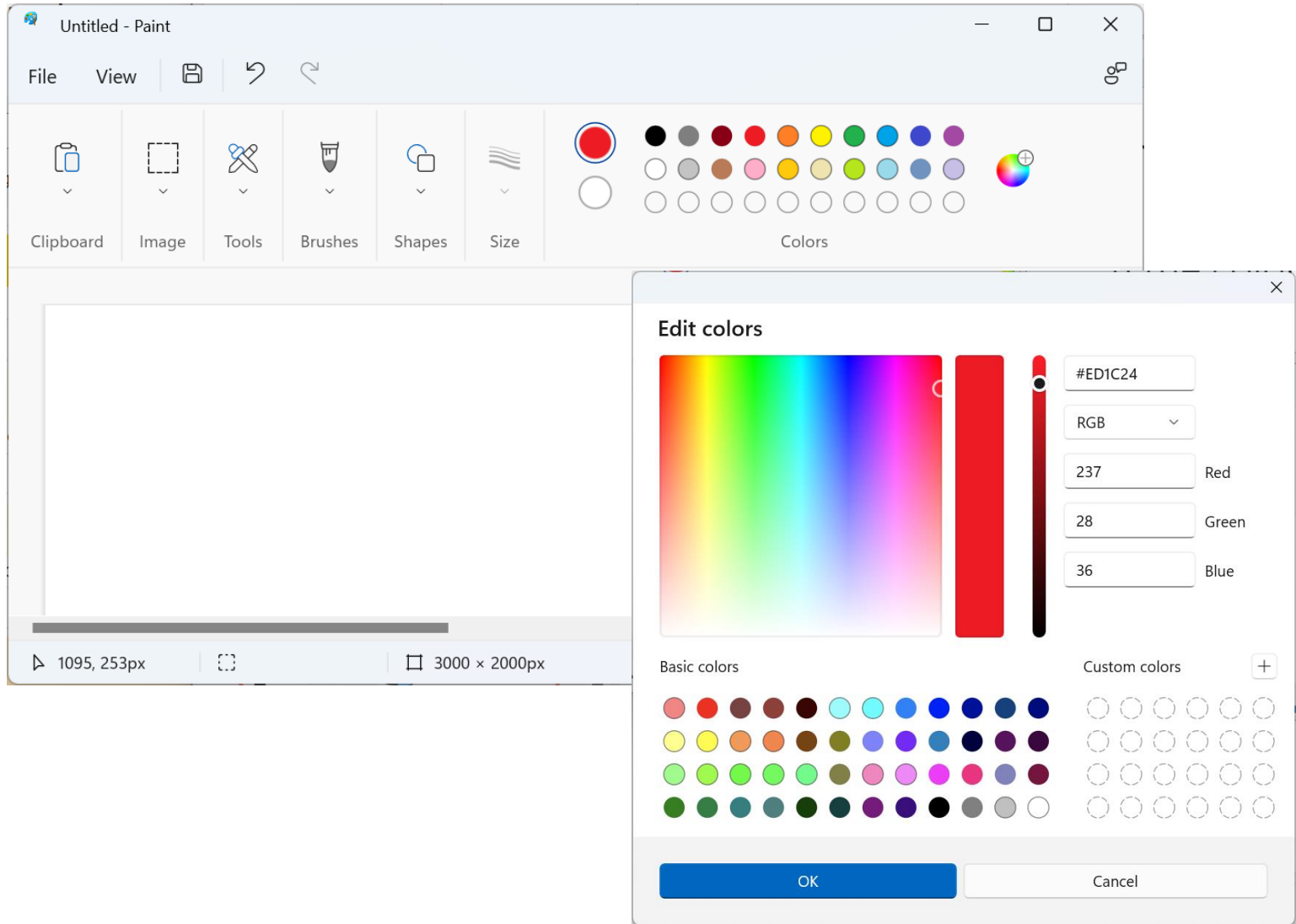


# Color palette

- Of course, you don't have to memorize all these named colors!
- You can find the entire list on the Backgrounds -> Colors palette
- Although they appear under Backgrounds, these are the same colors that you can also use for `stroke()` and `fill()` instructions



# What if the colors from codeguppy palette are not enough?



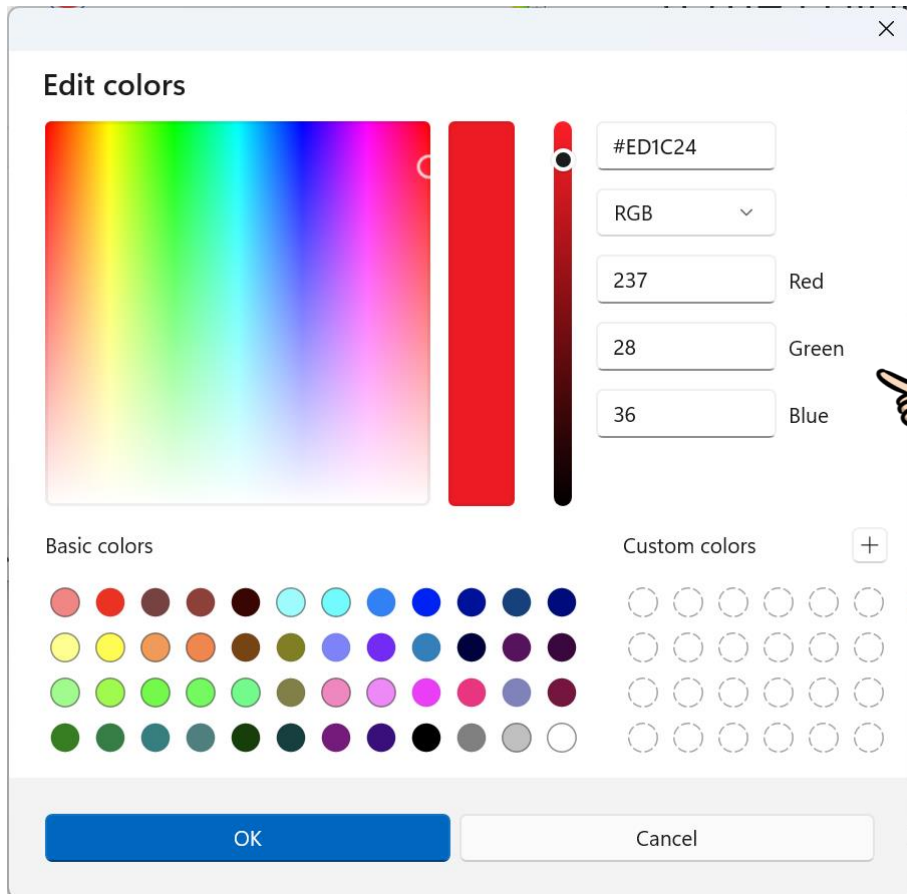
Did you ever use a drawing app such as Microsoft Paint?

Some apps allow users to select a wide range of colors and shades using a tool such as this one.



In JavaScript and codeguppy.com we can also use all these colors. Let's see how...

- Did you notice these fields called “Red”, “Green” and “Blue” in Microsoft Paint?  
(Note: They change automatically when you use the sliders to select other shades / hues.)
- In computing, **red**, **green** and **blue** are primary colors... and by combining these colors in different amounts we can obtain basically any other color or shade (or at least about 16 million other colors).




You can form any color by combining red, blue and green in different amounts in interval 0 ... 255

Red = 237  
Green = 28  
Blue = 36

Colors can also be expressed as a single text in hexadecimal format:

`"#ED1C24"`

# Using RGB colors in JavaScript

- Let's start by making a clone of the Flower program that we typed in earlier. Use the  button
- Using Microsoft Paint try to find similar RGB colors for the three colors that appear in the program. JavaScript and codeguppy allow defining colors in RGB format as well. You can specify them as 3 numbers or as a small text containing the hexadecimal code.

```
// Stem  
fill("lime");  
rect(277, 313, 30, 237);  
ellipse(215, 514, 124, 46);  
ellipse(374, 438, 134, 46 );
```

```
// Flower  
fill("red");  
circle(290, 160, 87);  
circle(209, 314, 87);  
circle(377, 307, 87);  
fill("yellow");  
circle(290, 260, 46);
```

```
// Stem  
fill(146, 208, 80);  
rect(277, 313, 30, 237);  
ellipse(215, 514, 124, 46);  
ellipse(374, 438, 134, 46 );
```

```
// Flower  
fill("#ED1C24");  
circle(290, 160, 87);  
circle(209, 314, 87);  
circle(377, 307, 87);  
fill("#FFFD55");  
circle(290, 260, 46);
```

# Filled shapes with no outline (no stroke)

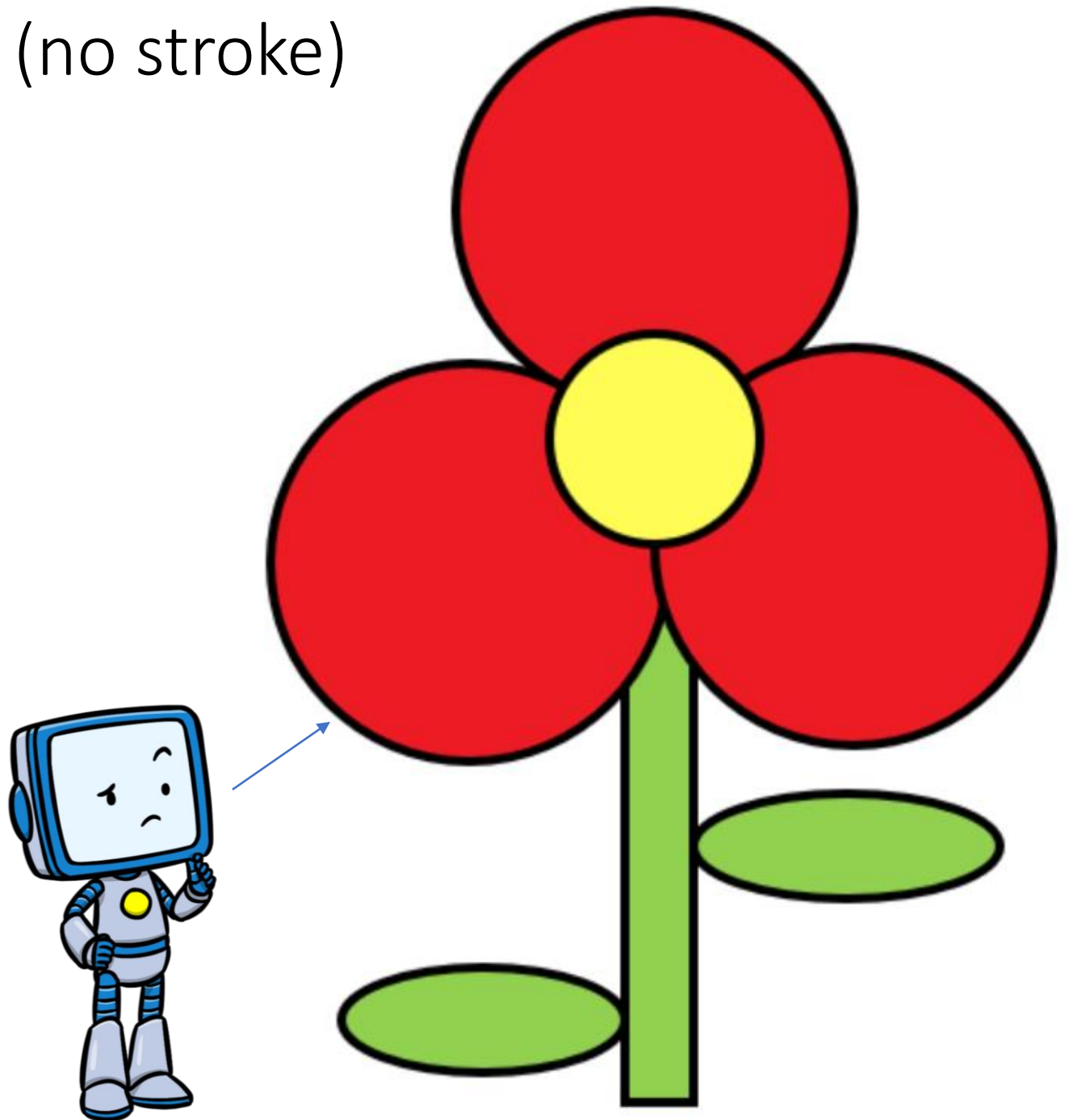
You probably noticed in the flow program, that the shapes that makes our flower have a tiny black outline. This is because of the default stroke used by codeguppy.

We also learned that we can change the color of the stroke using the `stroke()` command.

**Question:** But how can we eliminate the outline completely?

**Answer:** This can be done by using the `noStroke()`; instruction at the beginning of the program.

Go ahead, try to update your program with this instruction and see the effect (full listing on next page).



# Removing the outline...

```
noStroke();
```



```
// Stem
```

```
fill(146, 208, 80);
```

```
rect(277, 313, 30, 237);
```

```
ellipse(215, 514, 124, 46);
```

```
ellipse(374, 438, 134, 46 );
```

```
// Flower
```

```
fill("#ED1C24");
```

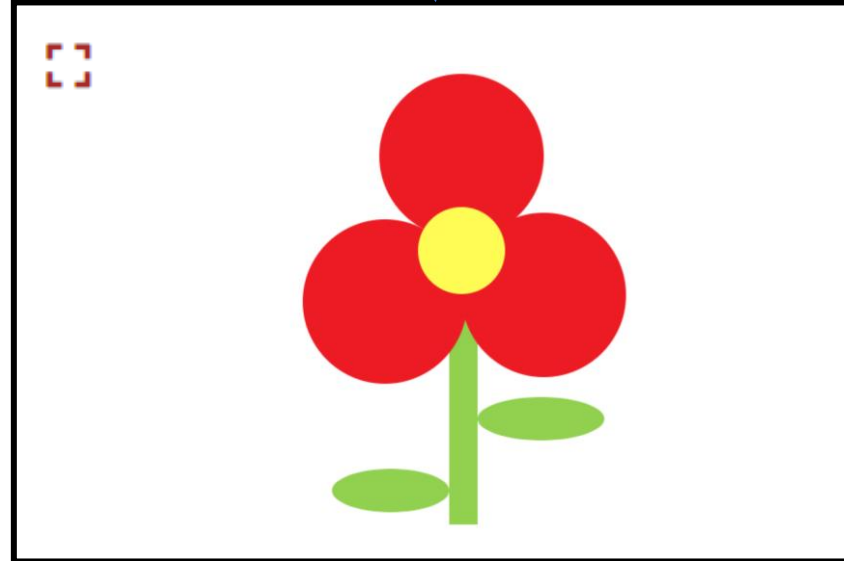
```
circle(290, 160, 87);
```

```
circle(209, 314, 87);
```

```
circle(377, 307, 87);
```

```
fill("#FFD55");
```

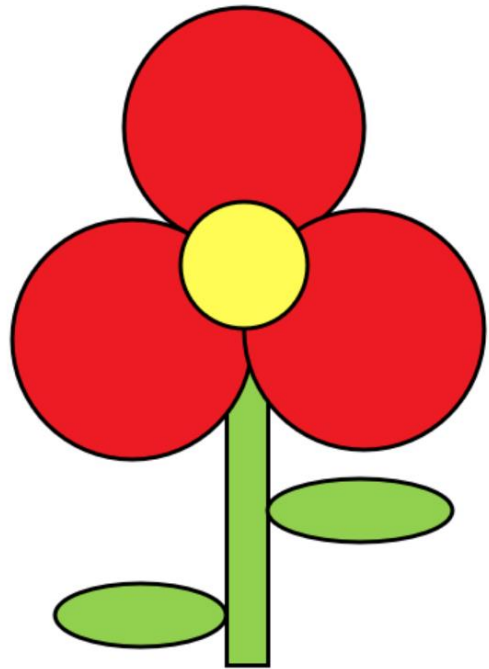
```
circle(290, 260, 46);
```



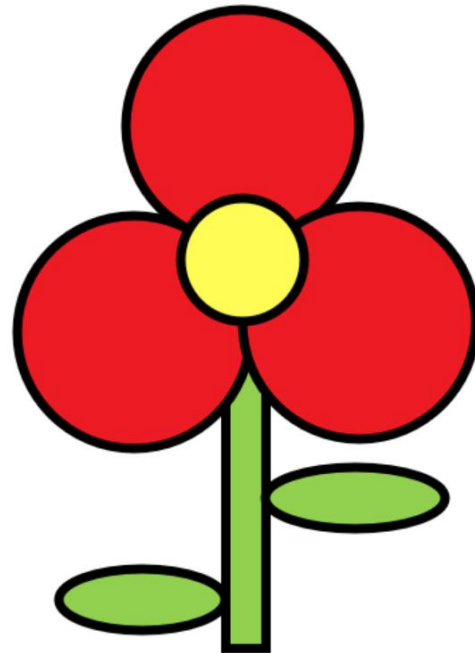
- Filled shapes with no outline!
- `noStroke();` will affect all the future outlines until a new outline color is selected via `stroke("red");`
- Note: "S" is uppercase inside `noStroke()` instruction. Also, `noStroke` is one word (with no space in between *no* and *Stroke*)

# Can you select a different thickness for strokes?

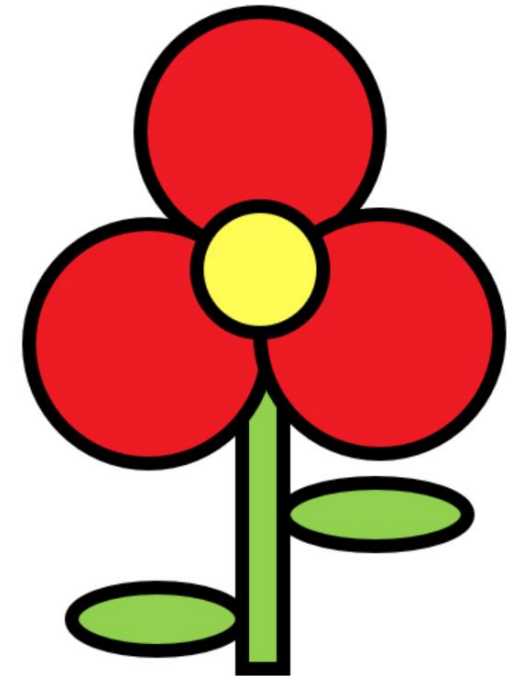
- You can use the `strokeWeight()` instruction to select a different thickness for the outline
- By default, the thickness is one, but you can use any other number like in this examples
- Modify the previous program and instead of `noStroke()` use `strokeWeight()` instruction as you see below



```
strokeWeight(3);
```



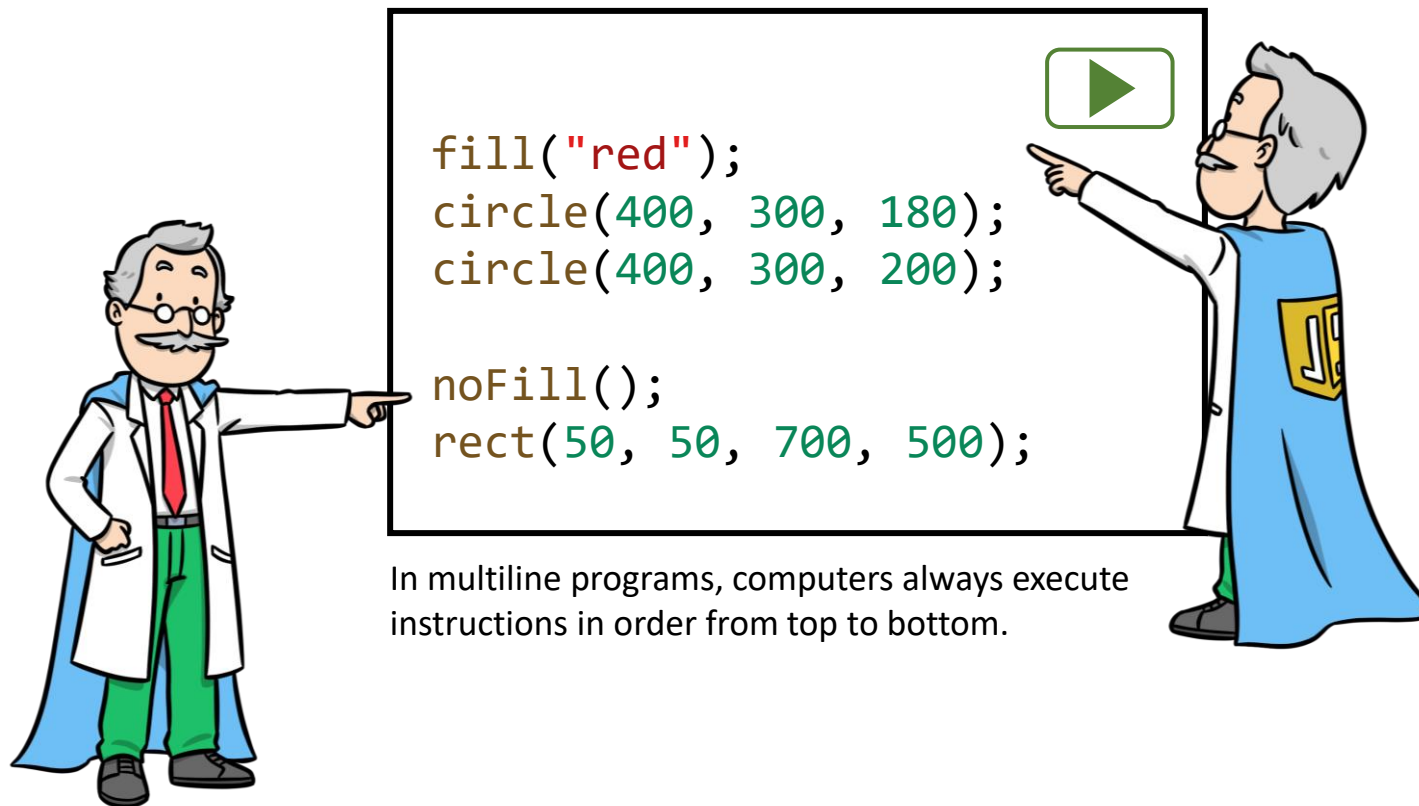
```
strokeWeight(7);
```



```
strokeWeight(10);
```

# You can also remove the fill color!

- You can use the `noFill()` instruction to remove the fill color
- By default, shapes in codeguppy environment are drawn without fill.
- In the following program the second circle, which is bigger will hide completely the first one. However, since the rectangle is drawn transparent (with no fill), will not hide the circles.



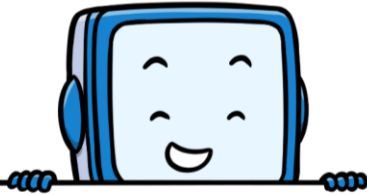
```
fill("red");  
circle(400, 300, 180);  
circle(400, 300, 200);  
  
noFill();  
rect(50, 50, 700, 500);
```

In multiline programs, computers always execute instructions in order from top to bottom.



# Text can be as big as you want

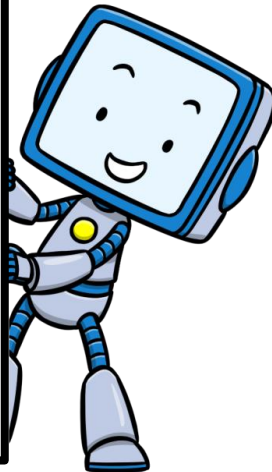
- To control the text size, you can use the `textSize()` instruction, with a number as parameter.
- Notice that the other attribute instructions such as `stroke()`, `strokeWeight()` and `fill()` are also affecting the text appearance.



```
textSize(120);  
  
stroke("blue");  
strokeWeight(7);  
fill("yellow");  
  
text("Hello", 250, 200);  
text("JavaScript", 150, 400);
```

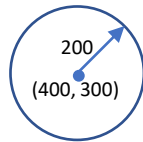


Hello  
JavaScript

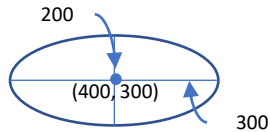


# Drawing Shapes

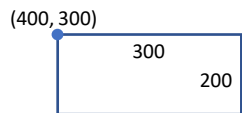
```
circle(400, 300, 200);
```



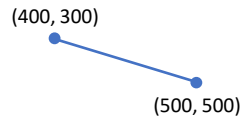
```
ellipse(400, 300, 300, 200);
```



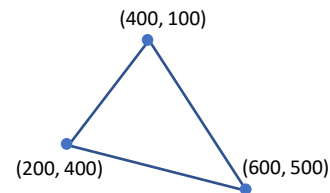
```
rect(400, 300, 300, 200);
```



```
line(400, 300, 500, 500);
```



```
triangle(400, 100, 200, 400, 600, 500);
```



```
arc(400, 300, 300, 200, 0, 180);
```



```
point(400, 300);
```



```
text('JavaScript', 400, 300);
```



# Setting shape attributes

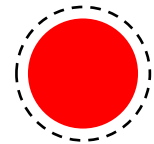
```
background("red");
```



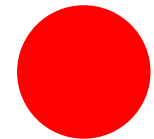
```
stroke("red");
```



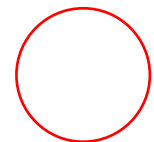
```
noStroke();
```



```
fill("red");
```



```
noFill();
```



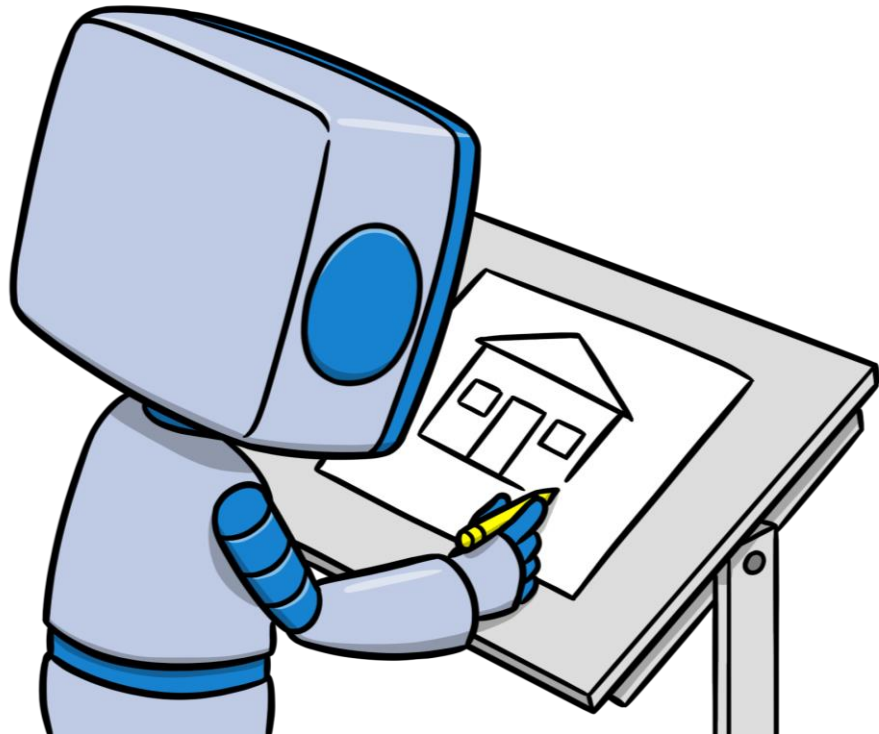
```
strokeWeight(3);
```



```
textSize(10);
```

HELLO

# Typing time! Let's type-in this program...



```
// Background
noStroke();
fill("#00b0f0");
rect(0, 0, 800, 400);
fill("#548235");
rect(0, 400, 800, 200);

fill("#ffc740");
rect(109, 254, 325, 274);
```

```
// Left window
stroke("#c55a11");
strokeWeight(3);
fill("#dae3f3");
square(147, 302, 56);
line(175, 302, 175, 358);
line(147, 330, 203, 330);
```

```
// Right window
square(347, 302, 56);
line(375, 302, 375, 358);
line(347, 330, 403, 330);
```

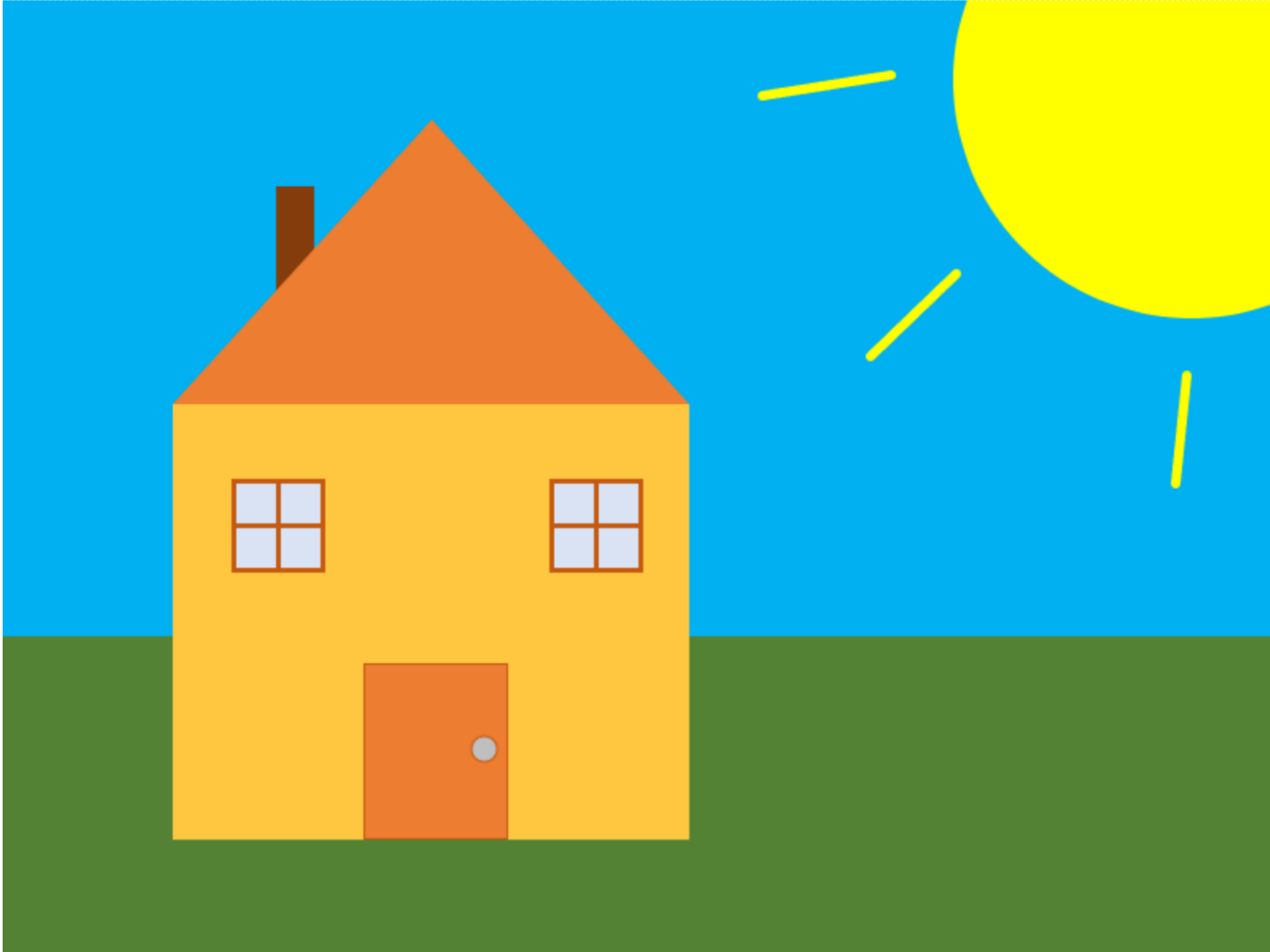
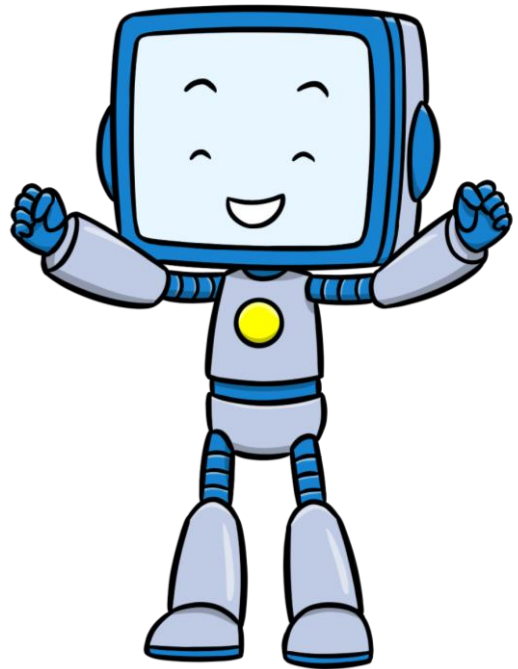
```
// Door
strokeWeight(1);
fill("#ed7d31");
rect(229, 417, 90, 110);
fill("#bfbfbf");
circle(305, 471, 8);
```

```
// Horn
noStroke();
fill("#843c0c");
rect(174, 117, 24, 80);
```

```
// Roof
fill("#ed7d31");
triangle(109, 254, 272, 75, 434, 254);
```

```
// Sun
fill("yellow");
circle(750, 50, 150);
stroke("yellow");
strokeWeight(6);
line(480, 60, 561, 47);
line(548, 224, 602, 172);
line(740, 304, 747, 236);
```

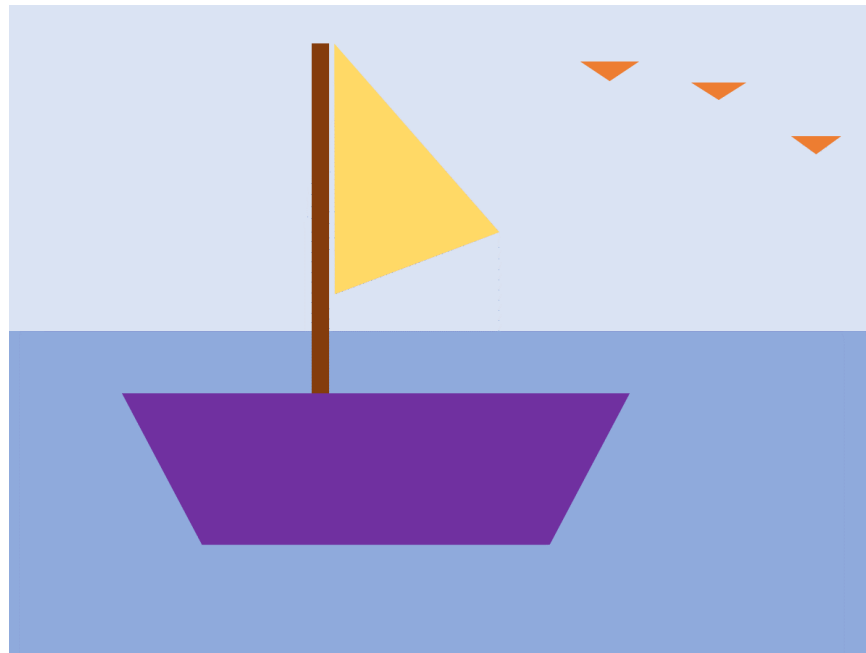
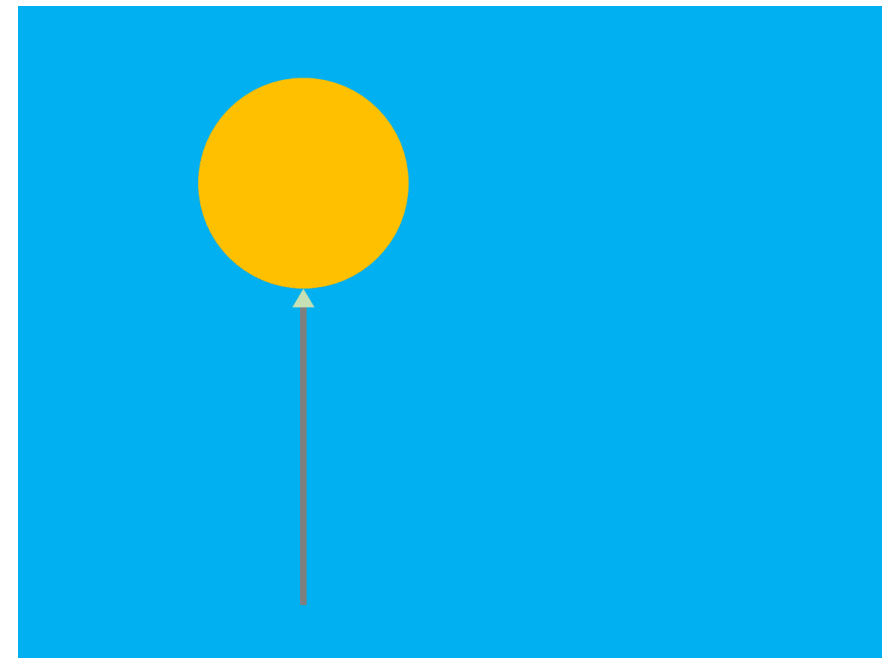
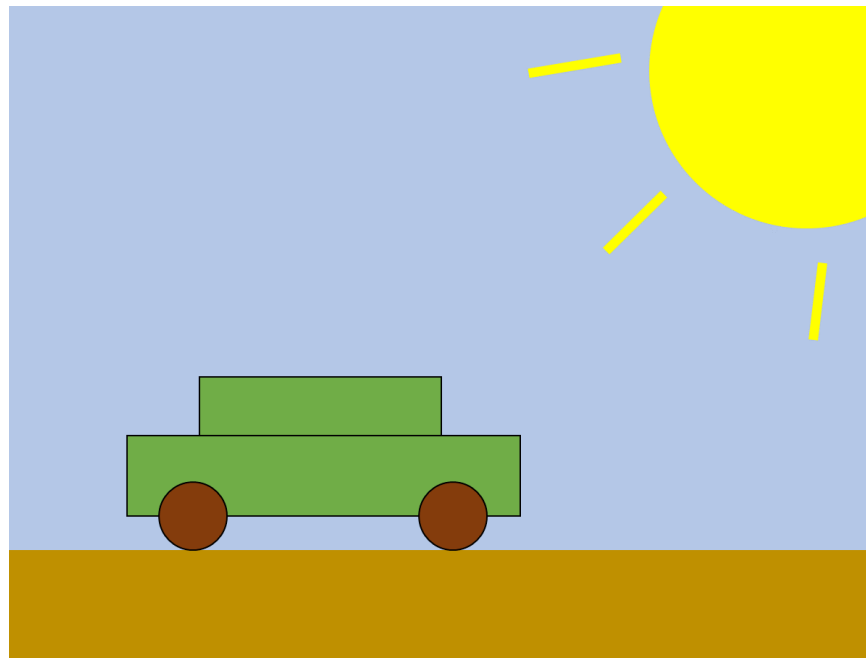
If you typed in correctly  
the program, you  
should see this image!



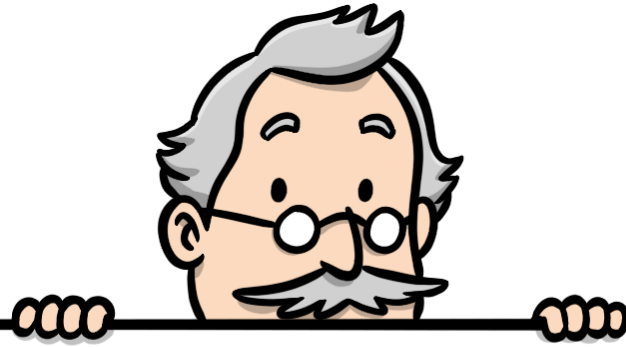
# Homework

Let's practice the new instructions by using them to draw a nice scene with code!

Here are a few examples, but feel free to create your own drawing.



Share your drawing with the entire class next time!



## Chapter V – Variables

- Printing messages using *println*
- Expressions
- Introducing variables
- **Exercise:** Calculate area and circumference of circle
- **Exercise:** Fahrenheit to Celsius converter
- **Exercise:** Population of Mars
- Incrementing variables
- Variables in graphical programs
- **Exercise:** Tiny house in the middle of the canvas
- **Exercise:** Tiny car at x, y coordinates

# Introducing *println* instruction

`println` is a simple instruction that can be used to quickly display values on the screen.

`println` is not using the canvas as the `text` instruction, but instead it displays the values in a simple text layer on top of the canvas.

```
println("Math program");  
println(100);  
println("+");  
println(200);  
println("=");  
println(300);
```

Text message

Number

```
Math program  
100  
+  
200  
=  
300
```

Notice that `println` is printing both numbers and texts.

If you want to print a number, type it as is, however if you want to print a message, please put it in between “quotes”

Feel free to explore `println` instruction by printing different numbers and messages on the screen.

You don't have to do calculations yourself!  
Computers are very efficient at doing math calculations.



Modify the last line in the program and let the computer help you with the calculation

```
println("Math program");  
println(100);  
println("+");  
println(200);  
println("=");  
println(300);
```

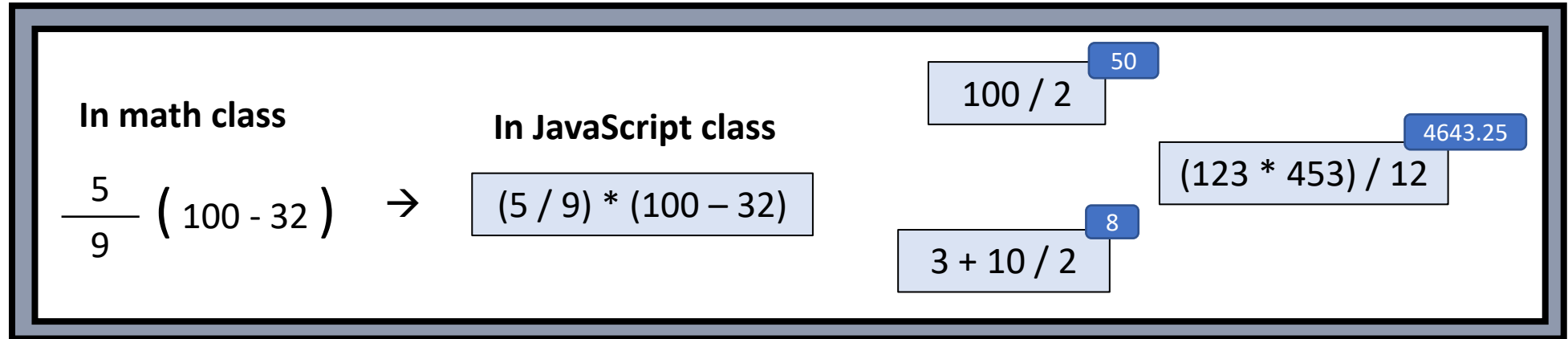


```
println("Math program");  
println(100);  
println("+");  
println(200);  
println("=");  
println(100 + 200);
```



# About expressions

In the previous example we provided JavaScript the expression `100 + 200`, and JavaScript performed the calculation. You can convert virtually any arithmetic expression to a JavaScript expression and let the computer calculate it.



In math class

$$\frac{5}{9} (100 - 32) \rightarrow$$

In JavaScript class

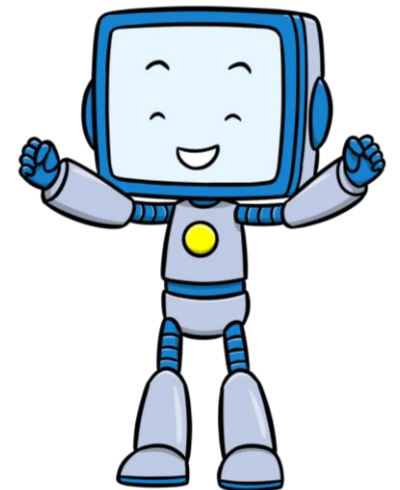
$$(5 / 9) * (100 - 32)$$

100 / 2 = 50

$$3 + 10 / 2 = 8$$
$$(123 * 453) / 12 = 4643.25$$

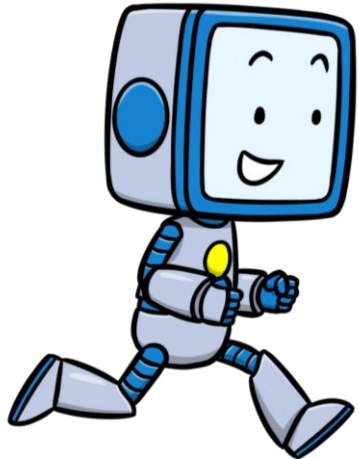

You can also use the following basic operators

Operator	Effect
+	Addition
-	Subtraction
*	Multiplication
/	Division
()	Parenthesis – set order of operations (if not used, JavaScript follows typical math order of operations)

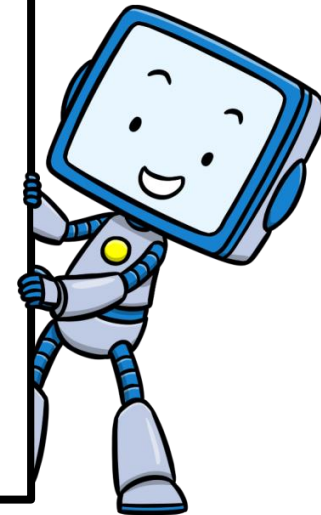



Practice expressions by doing as many calculations as needed.

You don't have to type in this program! This is just an example. Try to input your own calculations using the basic addition, subtraction, multiplication and division operators. Use parenthesis to indicate the desired order of operations.

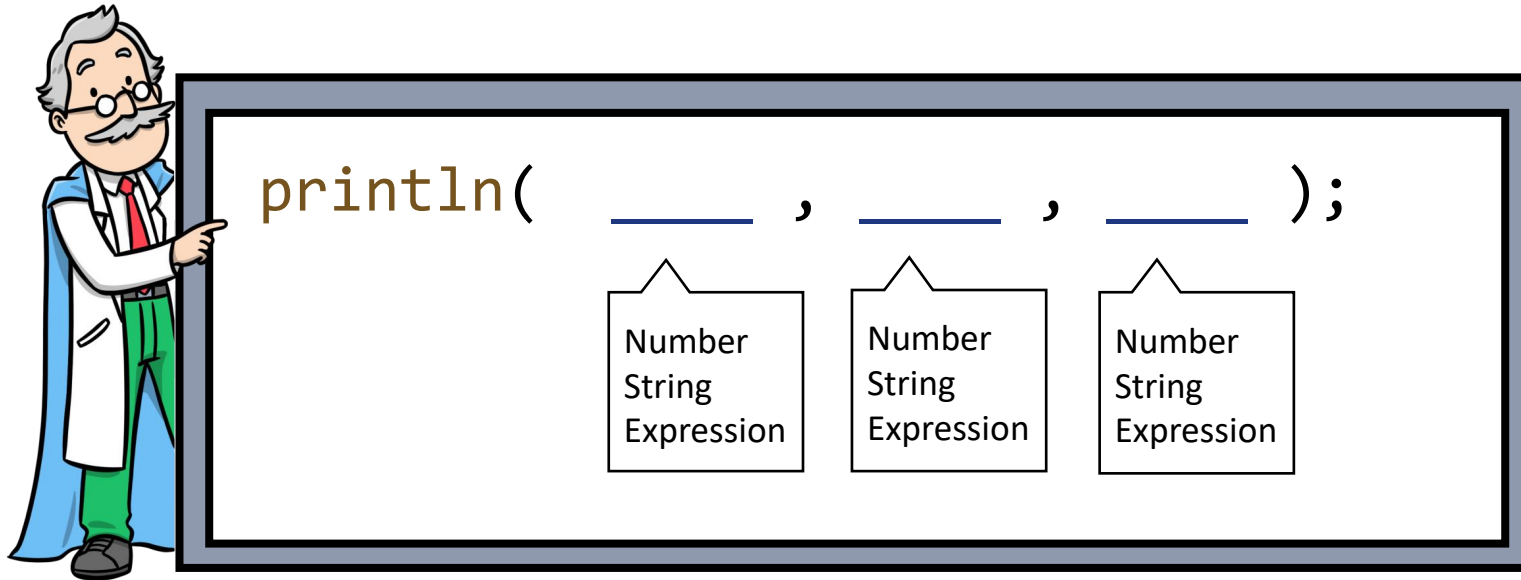


```
println("JavaScript calculator");  
  
println( 2 + 3 );  
  
println( 3 / 2 );  
  
println( 2 + 3 * 10 - 100 / 2 );  
  
println( 2 * (3 + 5) );  
  
println( ((5 - 3) * (5 - 3)) / 2 );  
  
println( (2 + 3) * 5 - 4 * (5 / 2) );
```



Type one `println` line at a time, then press Run. 

Tip: `println` accepts an unlimited number of arguments. When you provide multiple parameters, `println` will display all of them in a single line of text.



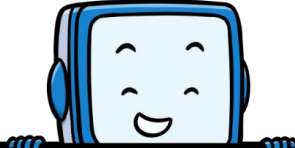
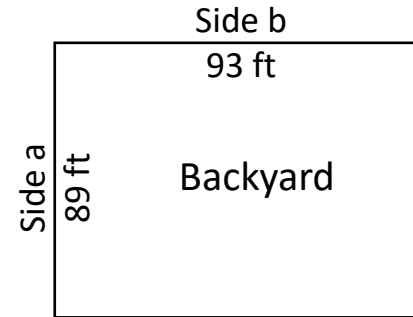
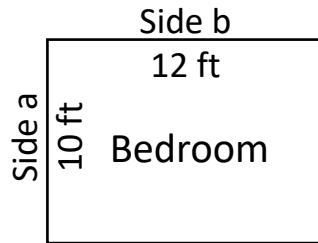
```
println("Calculation 1: ", 2 + 3 * 10 - 100 / 2);  
println("Calculation 2: ", 12 + 33 * 10 - 500 / 2);
```

The code block shows two lines of `println` statements. The first parameter of each statement is a string label, and the second parameter is a mathematical expression. Brackets above the code identify the string labels as "first parameter" and the mathematical expressions as "second parameter". A green play button icon is located in the top right corner of the code block.

You can use this feature to put a nice label in front of your calculations.


Sometimes, we need to execute the same calculations but with different numbers.

Let's say we need to calculate the perimeter and area of our bedroom and our backyard (both rectangles).



```
println("Side a: ", 10);
println("Side b: ", 12);

println("Perimeter = ", (10 + 12) * 2);
println("Area = ", 10 * 12);
```



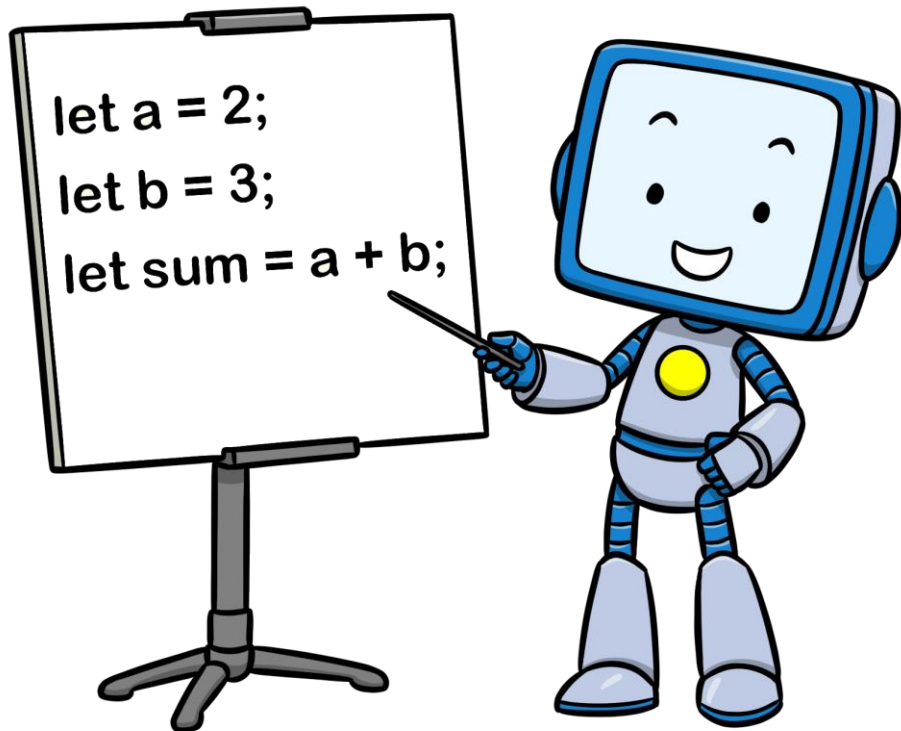
```
println("Side a: ", 89);
println("Side b: ", 93);

println("Perimeter = ", (89 + 93) * 2);
println("Area = ", 89 * 93);
```

It seems a lot of work to change the numbers in our programs each time we need to calculate the perimeter and area of a different rectangle. Is there a better way to minimize the changes?

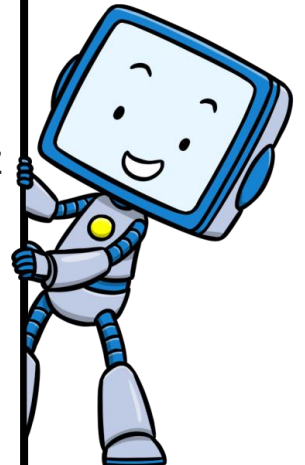
# Introducing variables

- JavaScript allows to define “variables”. A variable is a name that contains a value or expression.
- **let** keyword instructs the computer to define the variable with the specified name.
- Each time you use that name in your code, the computer will use the value represented by that name.



## Variable Rules

- You can use any letter or even combinations of letters and numbers for the name of the variable but don't start the variable name with a number or funny symbol
- The following are all valid variable names:  
a, b, c, ... , A, B, a1, b1, myName, houseSize, x, y, z
- Try to use meaningful letters or words for the names of the variables in order to refer to them easily at a later time. Don't use JavaScript reserved words.



let

name

=

value

;

a



10

b



3

c



100 / 2

result1



3 + 10 / 2

result2



12 + 33 \* 10 - 500 / 2

x



(9/5) \* 100 + 32

s



"Hello"

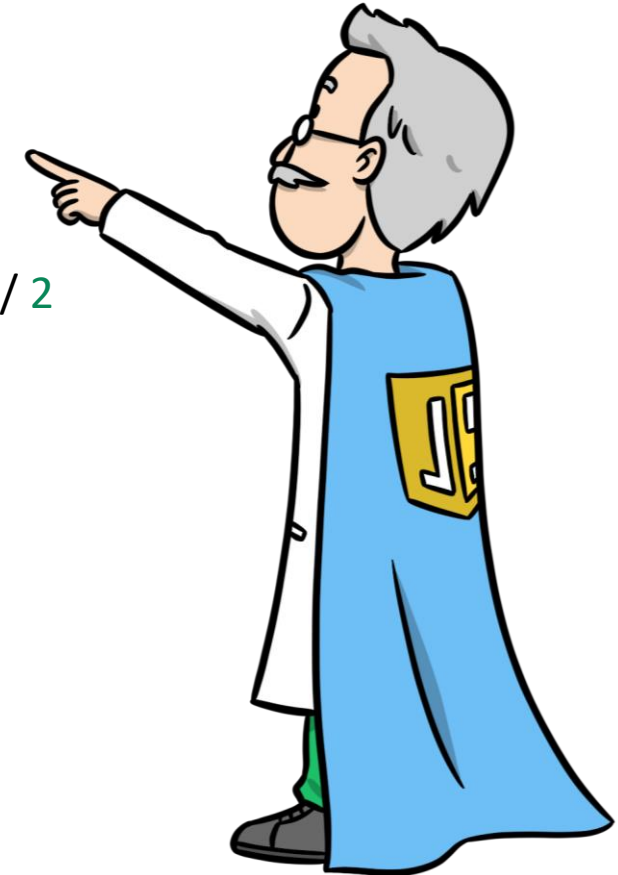
~~let~~

~~3c~~

As a value, you can assign a number, a text or an expression

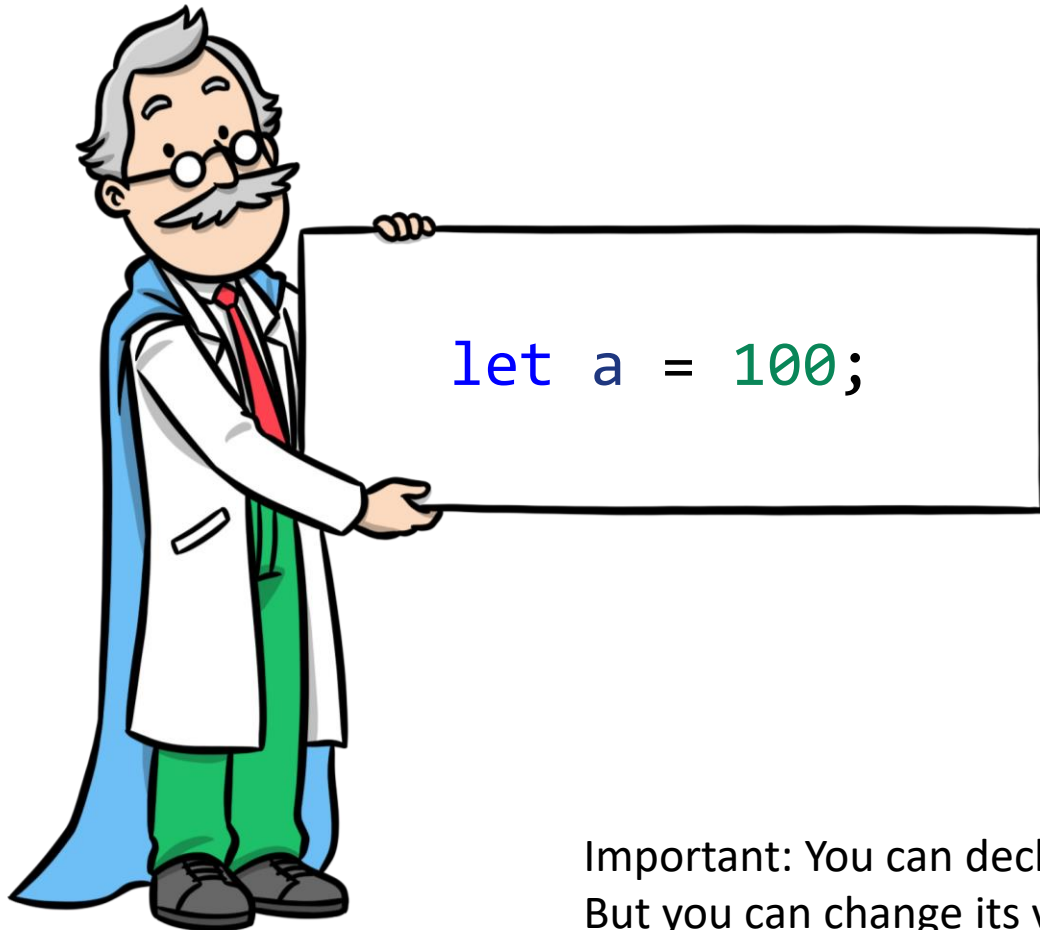


Put meaningful names to your variables following JavaScript rules



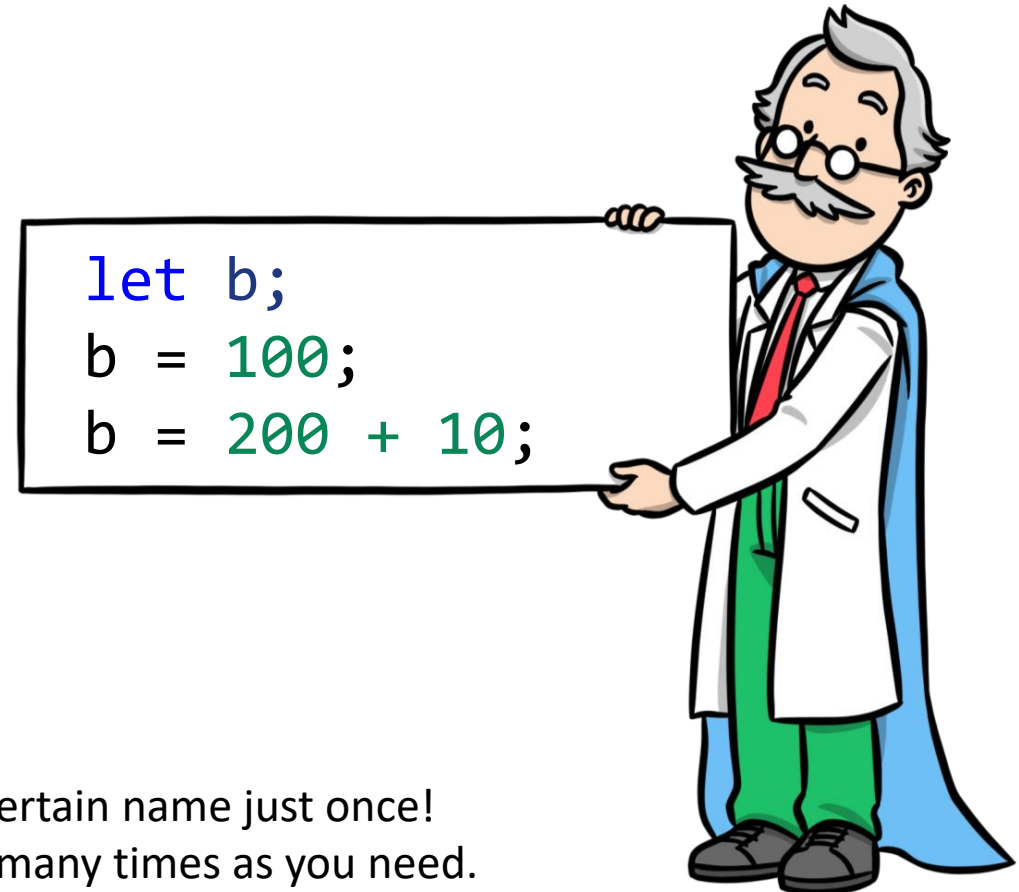
You can declare and assign variables in one line of code

Declare variable **a** and assign it with value **100**



Or you can just declare the variable ... then you can assign it later

1. Declare variable **b** (b has no value)
2. Assign variable **b** with value **100**
3. Reassign a new value to variable **b**. **b** is now **210**.



Important: You can declare a variable with certain name just once!  
But you can change its value (reassign it) as many times as you need.

If you want to inspect the value of a variable, you can use the same versatile `println` instruction.

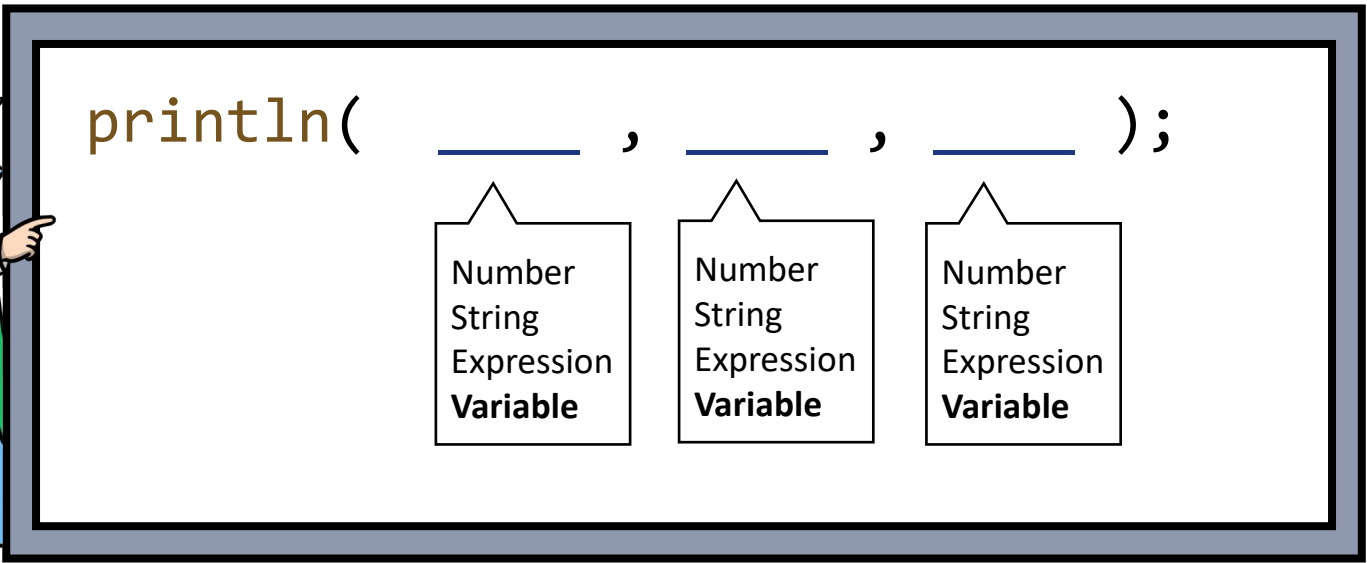
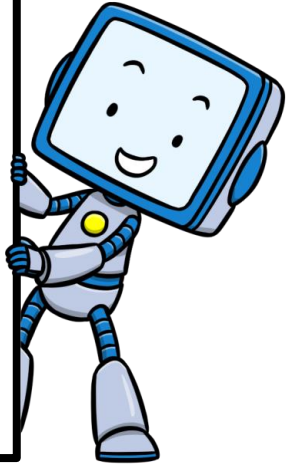
Program on the right shows a few ways of using `println` with variables.

```
let a = 10;
let b = 20;

println(a);
println(b);

println("a = ", a);
println("b = ", b);

println("a = ", a, " , b = ", b);
```



Remember: You can print multiple values in the same line using `println`.

As arguments you can use numbers, strings, expressions, and even variables.



Let's modify our room /rectangle calculation program to use two variables `a` and `b` to represents the rectangle sides.



```
let a = 10;  
let b = 12;  
  
println("Side a: ", a);  
println("Side b: ", b);  
println("Perimeter = ", (a + b) * 2);  
println("Area = ", a * b);
```



```
let a = 10;  
let b = 12;
```



```
Side a: 10  
Side b: 12  
Perimeter = 44  
Area = 120
```

```
let a = 89;  
let b = 93;
```



```
Side a: 89  
Side b: 93  
Perimeter = 364  
Area = 8277
```

By using variables, we can quickly re-run the program to calculate the size of different rooms / backyards / rectangles.

We only need to change the first two lines then press Run. 

```
let a = 23;  
let b = 19;
```



```
Side a: 23  
Side b: 19  
Perimeter = 84  
Area = 437
```

```
let a = 60;  
let b = 200;
```



```
Side a: 60  
Side b: 200  
Perimeter = 520  
Area = 12000
```

# String variables

Variables can store numbers as well as texts. If a variable stores a text, then we call that variable a “string variable”  
Numbers and strings are two important data types that are used by JavaScript.

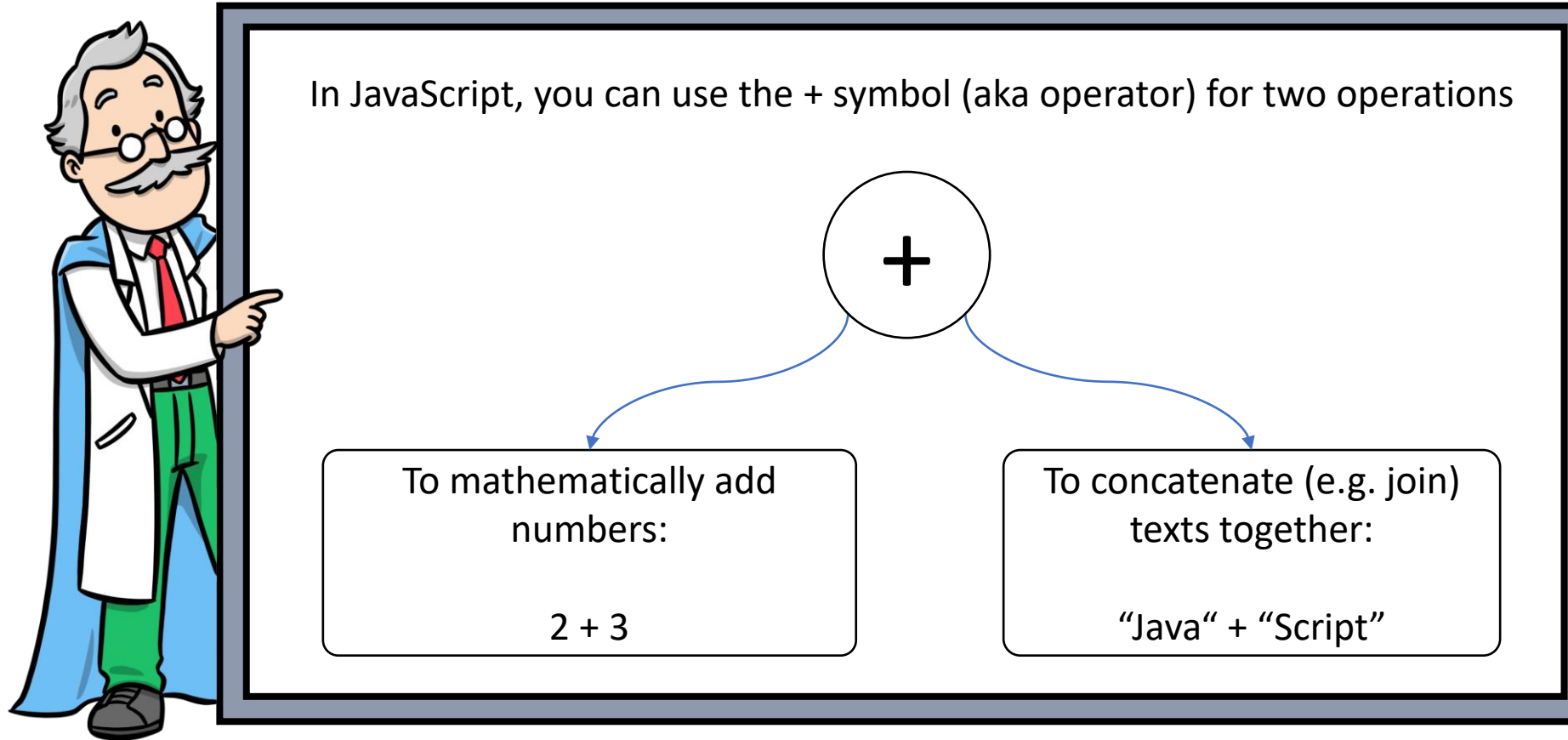
Variables can also store text values...

```
let s = "Hi";
```

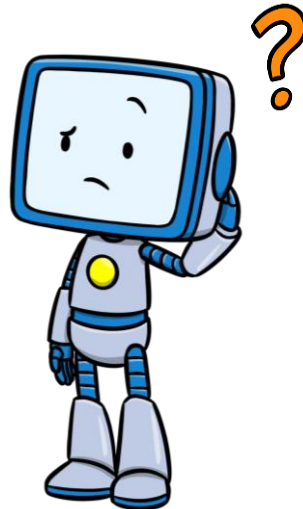
Strings can be concatenated using the “+” operator

```
let s1 = "Hello";  
let s2 = "World";  
  
let s = s1 + " " + s2 + "!";  
  
println(s);
```

# Concatenating strings



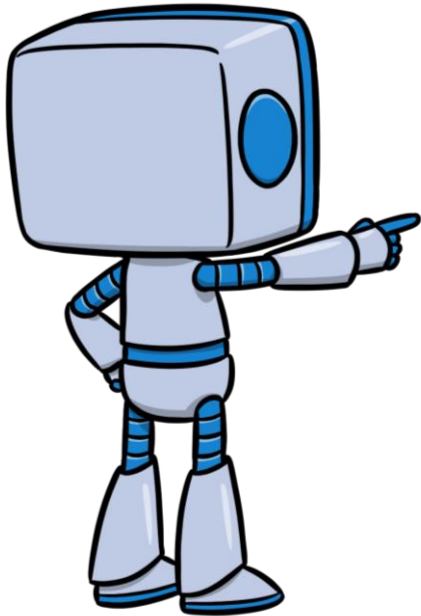
**Helpful tip:** If you try to concatenate a string with a *number*, JavaScript will first convert the number to *string* and then concatenate the two strings together.



# 100 vs "100"

When working with numbers, don't put them in between quotes if you intend to use them in mathematical expressions.

Although 100 and "100" looks the same, JavaScript interprets them differently. The first is a number, while the second is a string (e.g. a plain text).



```
let x1 = 100;
println( x1 + 3 );
```

103



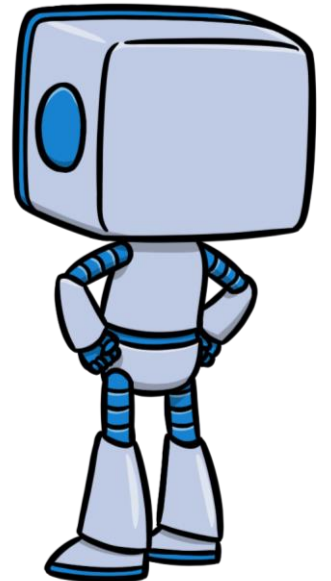
Since x1 is a number, the computer will perform the arithmetic and the output is 103 (100 + 3 = 103)

```
let x2 = "100";
println( x2 + 3 );
```

1003



Because x2 is a string, the computer will convert number 3 to string "3" and will *concatenate* the two strings outputting "1003"



```
let name = "Marian";
let myAge = 100;

println("My name is ", name, " and my age is ", myAge);
```



Until now we used *println* with multiple parameters to display different values

```
let name = "John";
let myAge = 100;

let message = "My name is " , name , " and my age is " , myAge;
println(message);
```



However, you cannot use the , to put multiple values together in a variable

(this program fails)

```
let name = "John";
let myAge = 100;

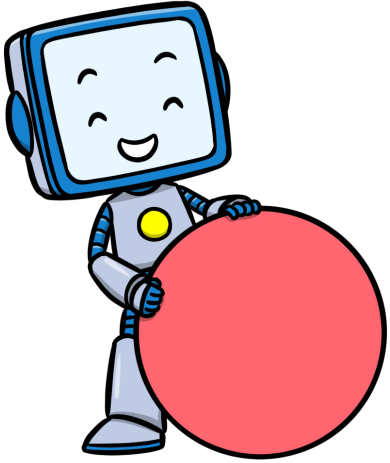
let message = "My name is " + name + " and my age is " + myAge;
println(message);
```



But you can use the + operator to concatenate together different values (number and text) and put the result in a variable!

# Exercise: Let's calculate area and circumference of a circle

This exercise requires a few basic notions of trigonometry.



For a circle of radius  $r$ , we can calculate the area  $A$  and circumference  $C$  with the following formulas

$$A = \pi * r^2$$

$$C = 2 * \pi * r$$

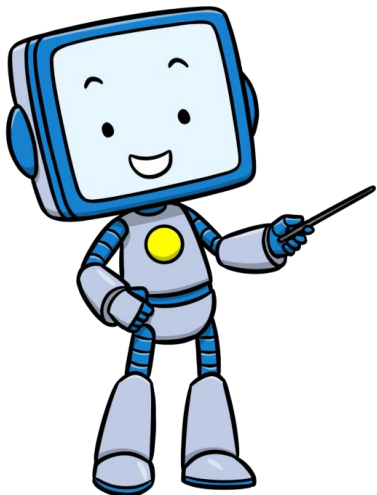
$$\pi = 3.14$$

We keep the radius in variable  $r$

codeguppy has a built-in constant named  $\text{PI}$  that has the value of  $\pi = 3.14$

Use variables ***area*** and ***circ*** for area and circumference of circle

Display the values



```
let r = 10;

let area = PI * r * r;
let circ = 2 * PI * r;

println("r = ", r);
println("Area = ", area);
println("Circumference = ", circ);
```



# Exercise: Fahrenheit to Celsius converter

## Fahrenheit to Celsius

$$T_C = \frac{5}{9} (T_F - 32) \quad \rightarrow (5 / 9) * (tf - 32)$$

## Celsius to Fahrenheit

$$T_F = \left( \frac{9}{5} T_C \right) + 32 \quad \rightarrow (9/5) * tc + 32$$

```
let tf = 100;
let tc = (5/9) * (tf - 32);
println(tf, " Fahrenheit = ", tc, " Celsius");

let tc2 = 100;
let tf2 = (9/5) * tc2 + 32;
println(tc2, " Celsius = ", tf2, " Fahrenheit");
```



Let's analyze the program:

- In the first part, we used variables **tc** and **tf** to hold the values for temperatures
- In the second part we used different variables **tc2** and **tf2** because we cannot declare the same variable twice – JavaScript rule! (we could've however not declared it second time, but only change value)
- We used parenthesis in expressions to dictate order of operations and to avoid ambiguity
- We used multiple parameters with **println** to give a nice format to our program output.

# Exercise: Population of Mars

In the future humans may colonize Mars.

**Let's find out how many people can inhabit Mars at the same population density as the people on Earth?**

Given data:

## Earth

Sphere with radius of 6378.1 km  
(use variable `earthRadius`)

Population of 7.753 billion people  
(use variable `earthPopulation`)

## Mars

Sphere with radius of 2106.9 miles  
(use variable `marsRadius`)

`marsPopulation = ?`

P.S. 1 mile = 1.60934 km



Take your time and try to implement this program! You can use as many variables as needed to store intermediate calculations. Solutions on the next slide!



```
let kmInAMile = 1.6;

let earthRadius = 6378.1; // in km
let earthPopulation = 7.753 * 1000000000; // people on Earth
let earthSurface = 4 * PI * earthRadius * earthRadius;
let earthDensity = earthPopulation / earthSurface;

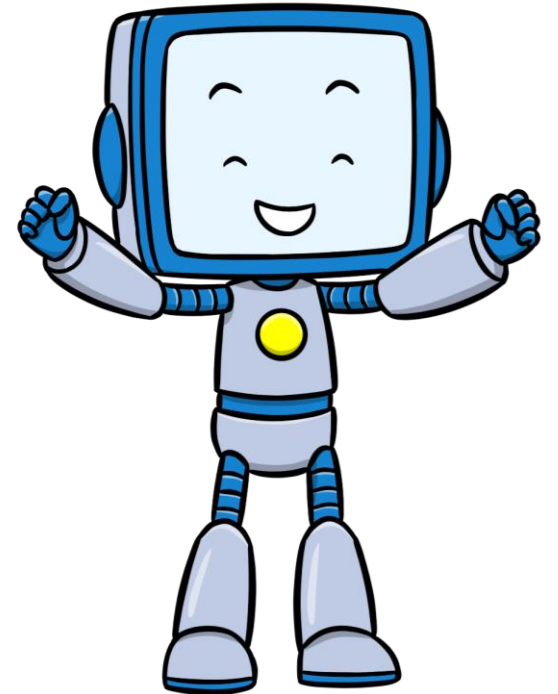
println("*** Earth ***");
println("Radius: ", earthRadius);
println("Surface: ", earthSurface);
println("Population: ", earthPopulation);
println("Density: ", earthDensity);

let marsRadius = 2106.1 * kmInAMile; // in km
let marsSurface = 4 * PI * marsRadius * marsRadius;
let marsPopulation = marsSurface * earthDensity;

println(" ----- ");
println("*** Mars ***");
println("Radius: ", marsRadius);
println("Surface: ", marsSurface);
println("Population: ", marsPopulation);
```

# Program listing

(Mars Population)



# Incrementing variables

Incrementing means increasing the value of a variable with a certain amount (very common is to increase with 1)

If  $a = 2$  and if we *increment*  $a$ , then  $a$  will become 3!

If  $x = 100$  and if we *increment*  $x$  by 10, then  $x$  will become 110

This is easy... we can write this code

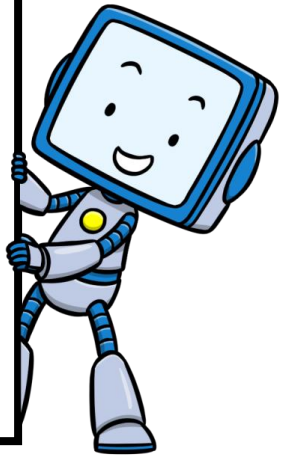
```
let a = 2;  
a = a + 1;  
  
let x = 100;  
x = x + 10;
```

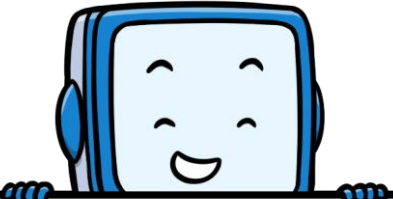
New value of x =

Old value of x + 10

The above code is correct, but incrementation is such a common operation in programming that in JavaScript it got its own ***incrementation*** operator.


We'll see it on the next slide.






```
let a = 2;
a = a + 1;

let x = 100;
x = x + 10;
```



```
let a = 2;
a++;

let x = 100;
x += 10;
```



```
let a = 2;
a--;
```

```
let x = 100;
x -= 10;
```

This code is equivalent. The one on the right is using the *incrementing* operators.

Did you know that JavaScript has also *decrementing* operators?

**++**      incrementing operator  
(increases value of variable by 1)

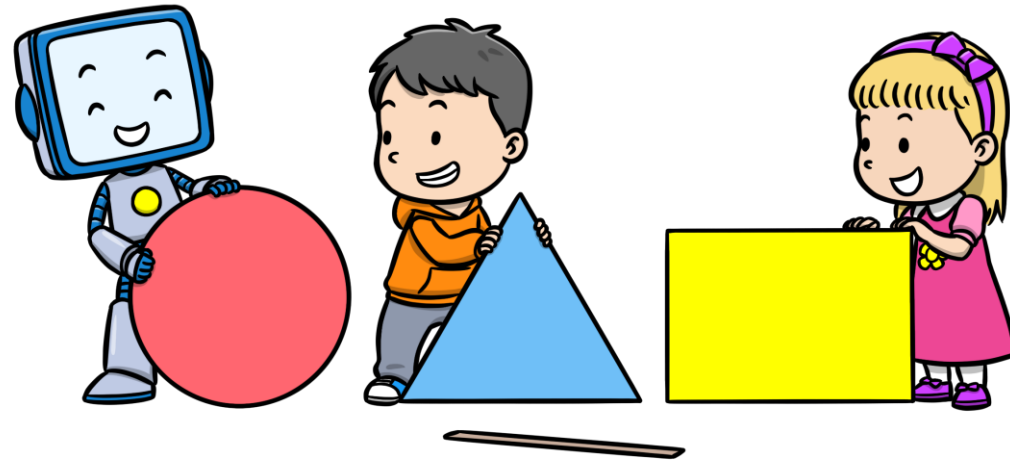
**+=**      incrementing operator  
(increases value of variable by specified amount)

**--**      decrementing operator  
(decreases value of variable by 1)

**-=**      decrementing operator  
(decreases value of variable by specified amount)



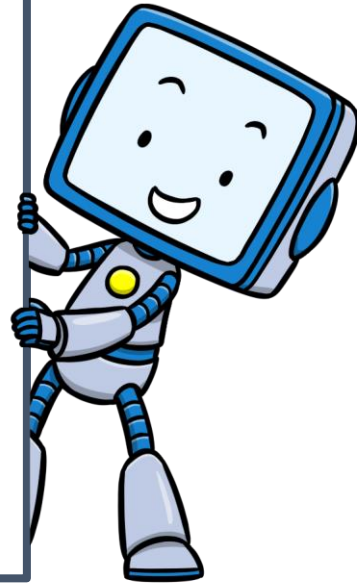
## Using variables in graphical programs



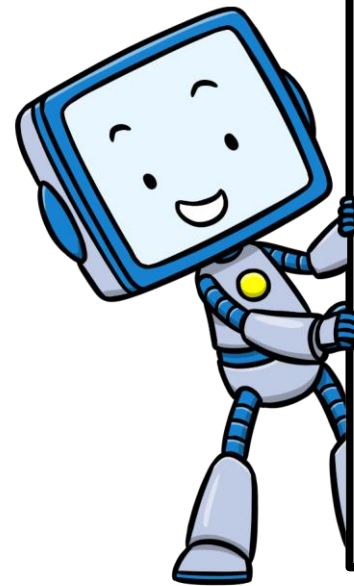
# Graphical instructions

Do you still remember the graphical instructions?

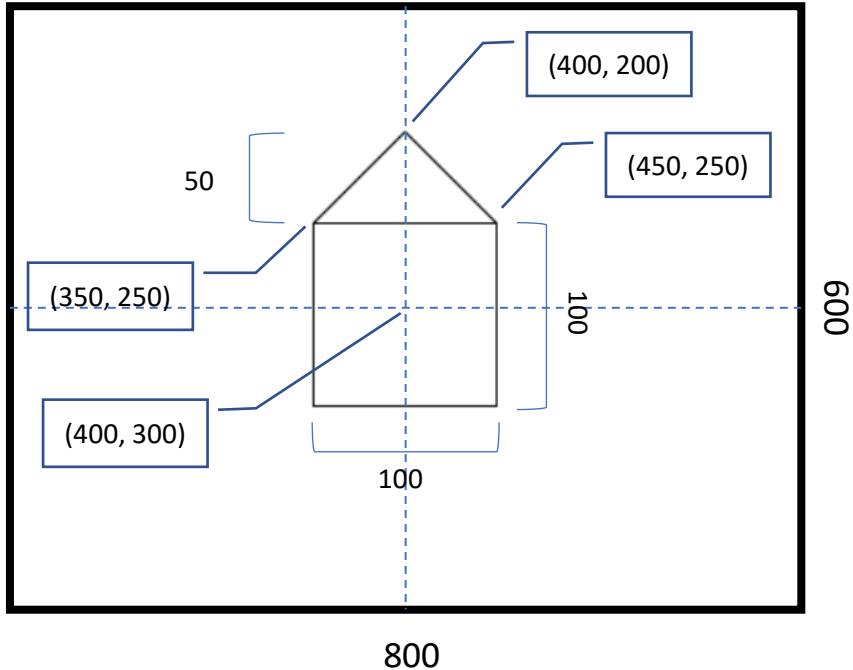
```
circle(400, 300, 200);  
ellipse(400, 300, 300, 200);  
rect(400, 300, 300, 200);  
line(400, 300, 500, 500);  
triangle(400, 100, 200, 400, 600, 500);  
arc(400, 300, 300, 200, 0, 180);  
point(400, 300);  
text('JavaScript', 400, 300);
```



```
background("red");  
stroke("red");  
noStroke();  
fill("red");  
noFill();  
strokeWeight(3);  
textSize(10);
```



# Exercise: Tiny house in the middle of the canvas



## Center of canvas

$$800 / 2 = 400 ; 600 / 2 = 300 \rightarrow (400, 300)$$

## Coordinates of the top-left corner of the square

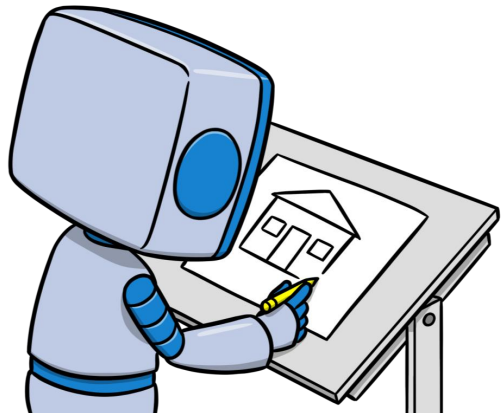
$$400 - 100 / 2 = 350 ; 300 - 100 / 2 = 250 \rightarrow (350, 250)$$

## Roof top

$$x = 400 ; y = 300 - 100 / 2 - 100 / 2 = 200 \rightarrow (400, 200)$$

## Coordinates of the top-right corner of the square

$$400 + 100 / 2 = 450 ; 300 - 100 / 2 = 250 \rightarrow (450, 250)$$



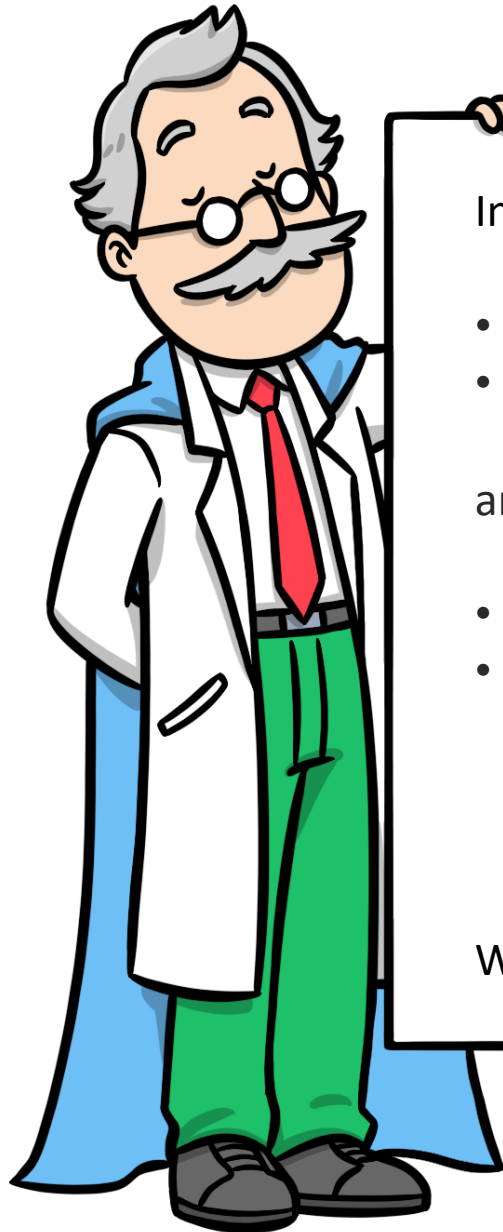
With calculations ready, it is trivial to draw the house using just 3 instructions!

Try the program. Press 

```
rect(350, 250, 450, 250);  
line(350, 250, 400, 200);  
line(400, 200, 450, 250);
```



# Let the computer calculate



In the tiny house program, try to replace:

- 350 with  $400 - 100 / 2$
- 250 with  $300 - 100 / 2$

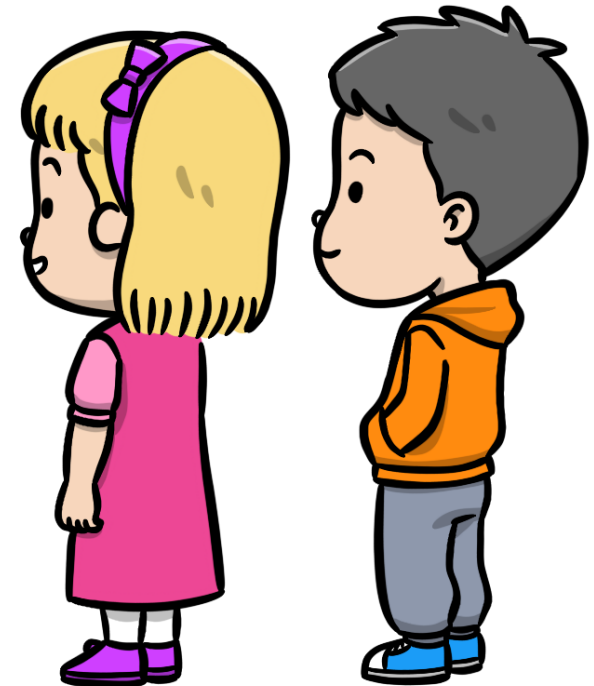
and

- 450 with  $400 + 100 / 2$
- 200 with  $300 - 100 / 2 - 100 / 2$

When ready, Press Run 

We will replace the specified numbers with expressions, so we let the computer find out the answers!

(don't use variables yet)



The code should look like this...

... and should draw the same tiny house!



Rectangle  
coordinates

Width and Height



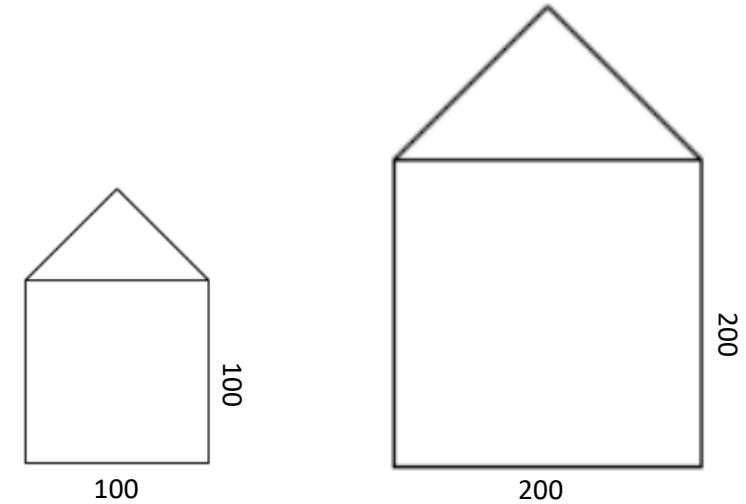
```
rect(400 - 100 / 2, 300 - 100 / 2, 100, 100);  
line(400 - 100 / 2, 300 - 100 / 2, 400, 300 - 100 / 2 - 100 / 2);  
line(400, 300 - 100 / 2 - 100 / 2, 400 + 100 / 2, 300 - 100 / 2);
```

Roof is half the size  
of the house



# A bigger house...

- Let's suppose we changed our mind, and we need to draw a bigger house. Instead of 100 pixels wide by 100 pixels height, our main square should be 200 by 200 pixels.
- Since our code contains the expressions that do the calculations, we only need to replace 100 with 200 wherever is needed.



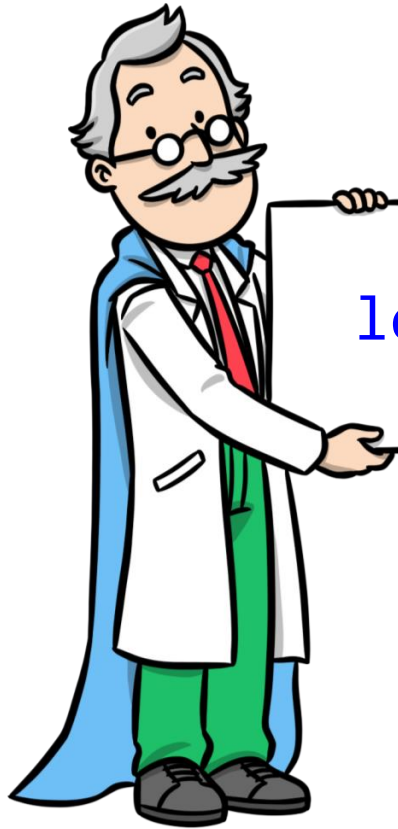
```
rect(400 - 200 / 2, 300 - 200 / 2, 200, 200);  
line(400 - 200 / 2, 300 - 200 / 2, 400, 300 - 200 / 2 - 200 / 2);  
line(400, 300 - 200 / 2 - 200 / 2, 400 + 200 / 2, 300 - 200 / 2);
```



- Due to expressions ... with just 12 replacements in code, we *asked* the computer to recalculate the new coordinates
- Let's see how to eliminate even these replacements when we'll be asked next time to change the size of our house...

# Adding variables to our program

- Let's clone the previous program and add variable  $h$  to the program (to keep the height of the house)
- "h" is used instead of the 100 or 200 values that we used before.



```
let h = 100;
```

When ready, Press Run

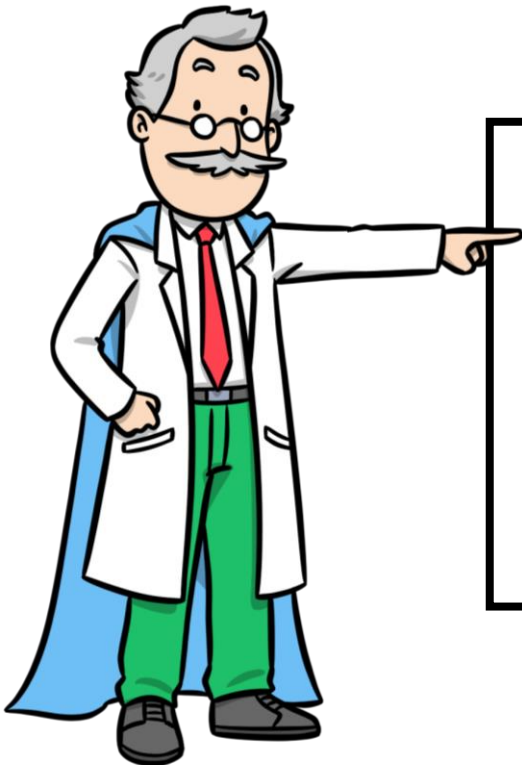
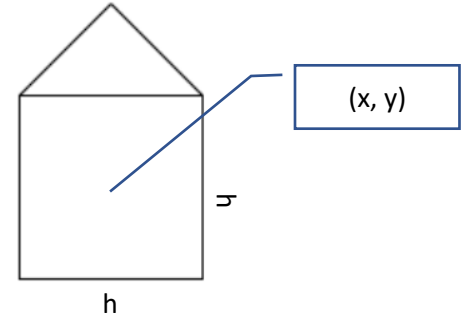
```
let h = 100;
```

```
rect(400 - h / 2, 300 - h / 2, h, h);  
line(400 - h / 2, 300 - h / 2, 400, 300 - h / 2 - h / 2);  
line(400, 300 - h / 2 - h / 2, 400 + h / 2, 300 - h / 2);
```



# Multiple variables

- Our graphical programs are not limited to just 1 variable. As a matter of fact, we can define practically an unlimited number of variables.
- Let's define two additional variables named x and y that will hold the coordinates where we want to draw our mini house.



```
let x = 400;  
let y = 300;  
let h = 100;
```

```
rect(x - h / 2, y - h / 2, h, h);  
line(x - h / 2, y - h / 2, x, y - h / 2 - h / 2);  
line(x, y - h / 2 - h / 2, x + h / 2, y - h / 2);
```



Try to complete this task on your own.

If you encounter difficulties refer to the code on the screen.

After you finish, play with the variables selecting other values.

Having fun with variables  
in this graphical program...



```
// Declare and initialize variables
let x = 400;
let y = 300;
let h = 100;

// Change the value of x by reassign it
x = 100;

rect(x - h / 2, y - h / 2, h, h);
line(x - h / 2, y - h / 2, x, y - h / 2 - h / 2);
line(x, y - h / 2 - h / 2, x + h / 2, y - h / 2);
```

Clone the program and add this `x = 100;` line  
This line reassign variable x to 100

Observe where the house is drawn.

```
// Just declare variables
let x;
let y;
let h;

// Then assign values to variables
x = 200;
y = 350;
h = 150;

rect(x - h / 2, y - h / 2, h, h);
line(x - h / 2, y - h / 2, x, y - h / 2 - h / 2);
line(x, y - h / 2 - h / 2, x + h / 2, y - h / 2);
```

Clone again the program and modify it as on the screen  
Now the `let` lines are only declaring the variables, and they  
are assigned later-on.

What do you think is happening if you forget to add the  
assignment lines?

# Exercise: Tiny car at x, y coordinates

Write a small program to draw a tiny car at x and y coordinates.

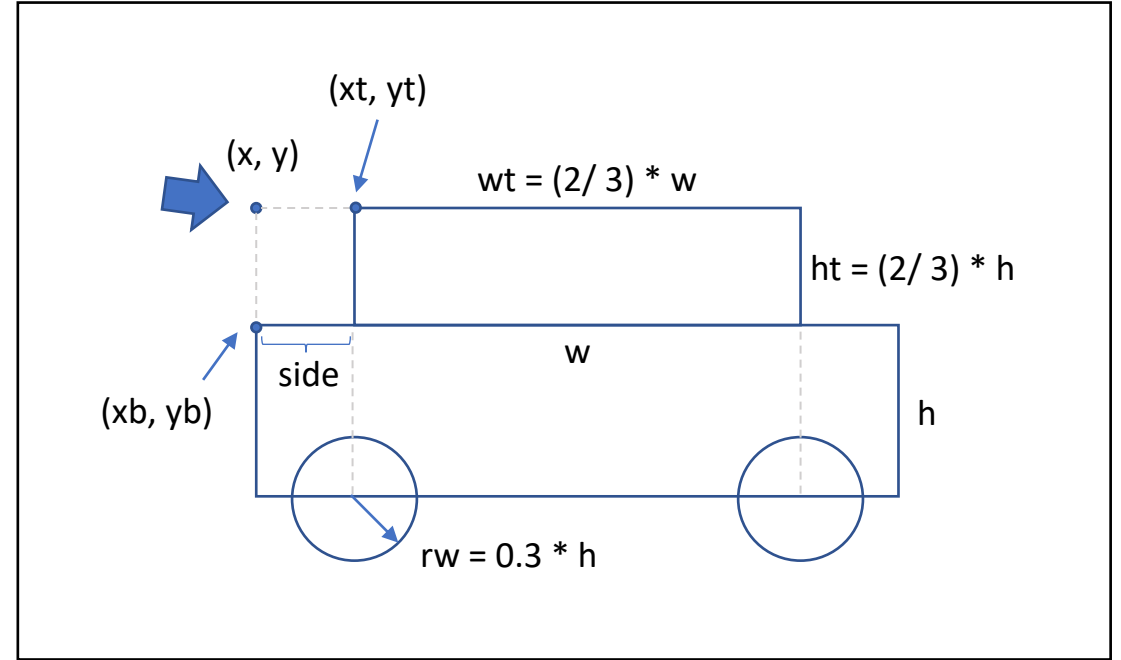
By changing the values of x and y and re-running the program, the car should be displayed at different positions.

Assume body width is 200 and body height is 50. See also drawing on the right with other calculations.

```
let x = 100;  
let y = 100;  
  
const w = 200;  
const h = 50;  
...
```



Tip: If value of a variable never changes, we can use keyword `const` instead of `let`.



## Car body calculations

$$xt = \frac{(w - wt)}{2} + x \quad yt = y$$
$$xb = x \quad yb = y + ht$$

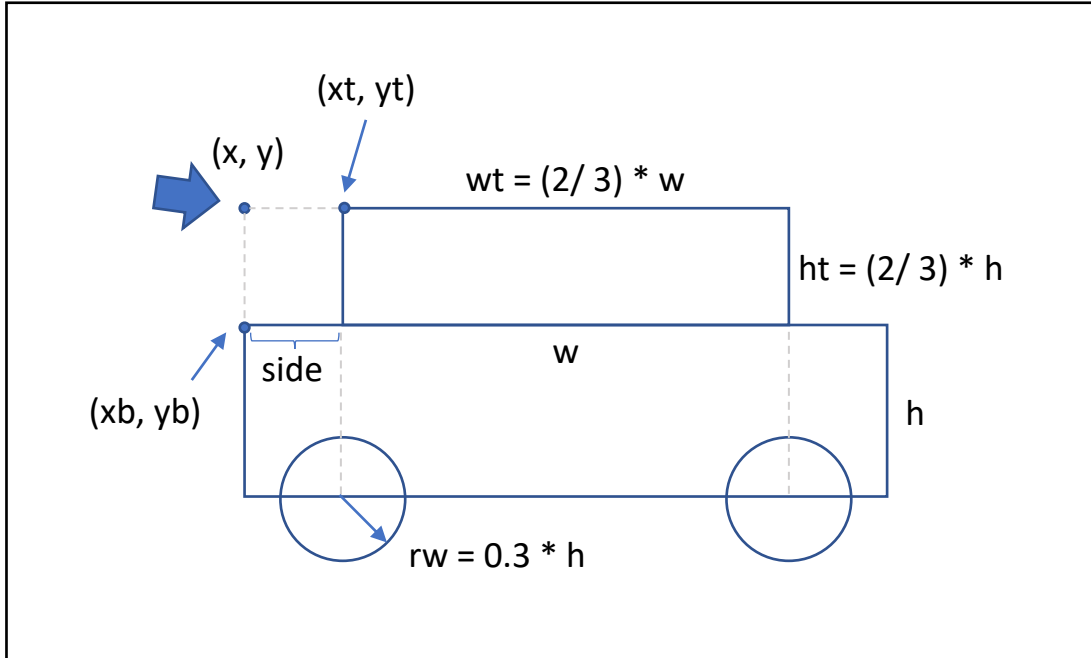
side

## Wheels calculations

$$yw = y + ht + h$$
$$xw1 = x + \frac{(w - wt)}{2}$$
$$xw2 = x + w - \frac{(w - wt)}{2}$$

Solution on next slide...

# Tiny car at x, y coordinates



```
let x = 100;  
let y = 100;  
  
const w = 200;  
const h = 50;  
const rp = 0.3;
```

```
// Calculate the width and height of top part  
let wt = (2 / 3) * w;  
let ht = (2 / 3) * h;
```

```
// Coordinates of top rectangle  
let side = (w - wt) / 2;  
let xt = x + side;  
let yt = y;
```

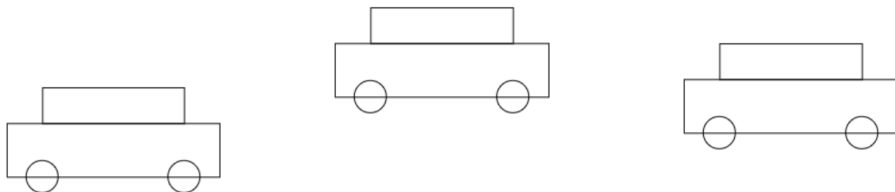
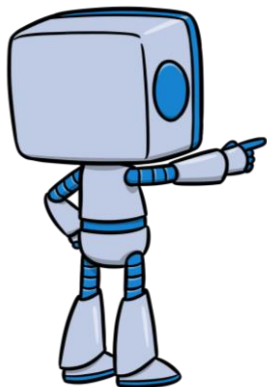
```
// Coordinates of bottom rectangle  
let xb = x;  
let yb = y + ht;
```

```
// Coordinates of wheels  
let xw1 = x + side;  
let xw2 = x + w - side;  
let yw = y + ht + h;  
let rw = h * rp;
```

```
rect(xt, yt, wt, ht);  
rect(xb, yb, w, h);  
circle(xw1, yw, rw);  
circle(xw2, yw, rw);
```

This is our version of program. How is yours?

Try changing the values of  $x$  and  $y$  and re-run the program. Do you see the car in different positions?

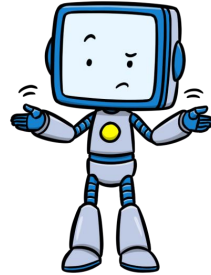


# Variables and Constants

In the previous exercises we used several times *constants*. Let's see what are constants in JavaScript.

## Variables

```
let x = 100;  
let y = 100;
```



## Constants

```
const w = 200;  
const h = 50;
```

In JavaScript, we define *variables* using keyword `let`, each time we know the value of that variable may change. If we know that the value shouldn't change, it is recommended to use a `const` keyword in declaration.

## Predefined constants

codeguppy.com provide some predefined constants. You can use them in any program you build.

```
println(PI);  
println(width);  
println(height);
```



**PI** contains the value of mathematical constant  $\pi = 3.14...$

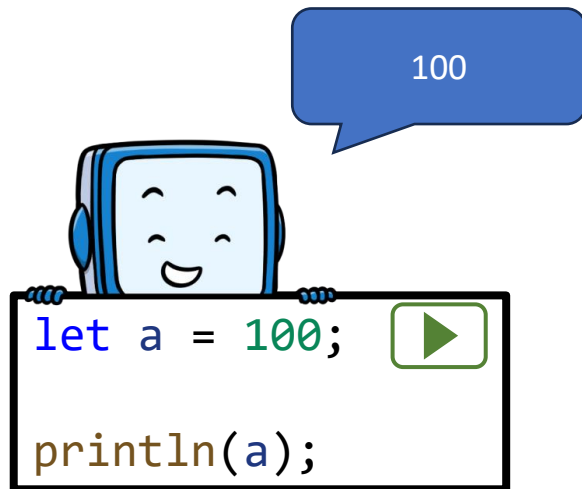
**width** = 800 (the width of the canvas)

**height** = 600 (the height of the canvas)

# Watch for: Uninitialized variables

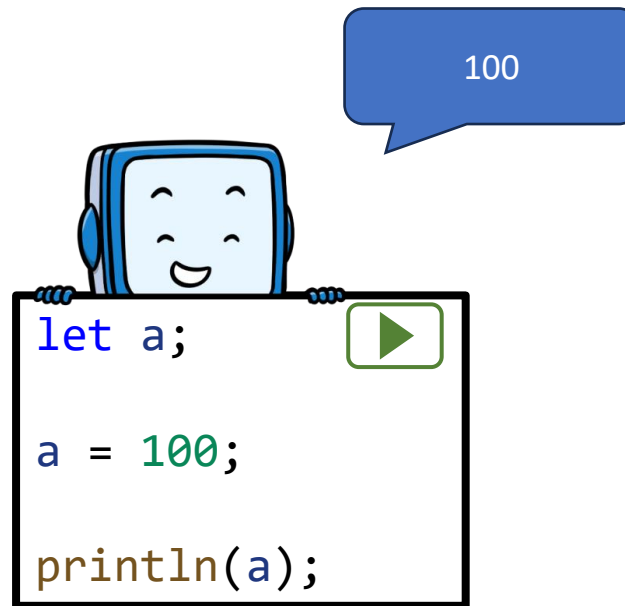
When you declare a variable in your program, JavaScript doesn't assign any value to it. The variable will remain in an "undefined" state until you assign a value.

Always initialize your variables. Uninitialized variables can be a source of errors in your programs.



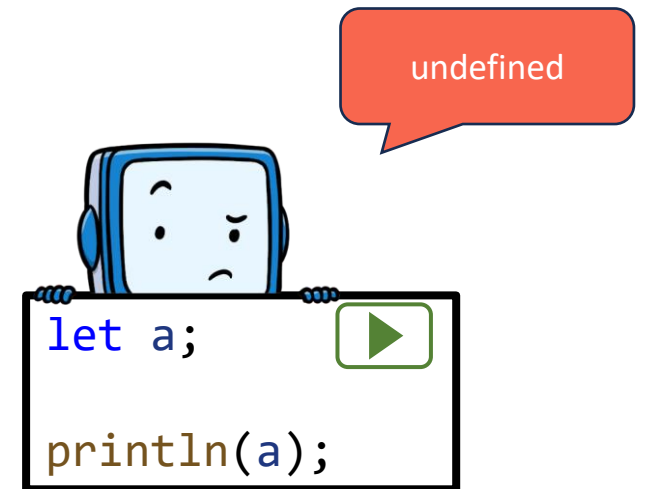
```
let a = 100;
println(a);
```

100



```
let a;
a = 100;
println(a);
```

100



```
let a;
println(a);
```

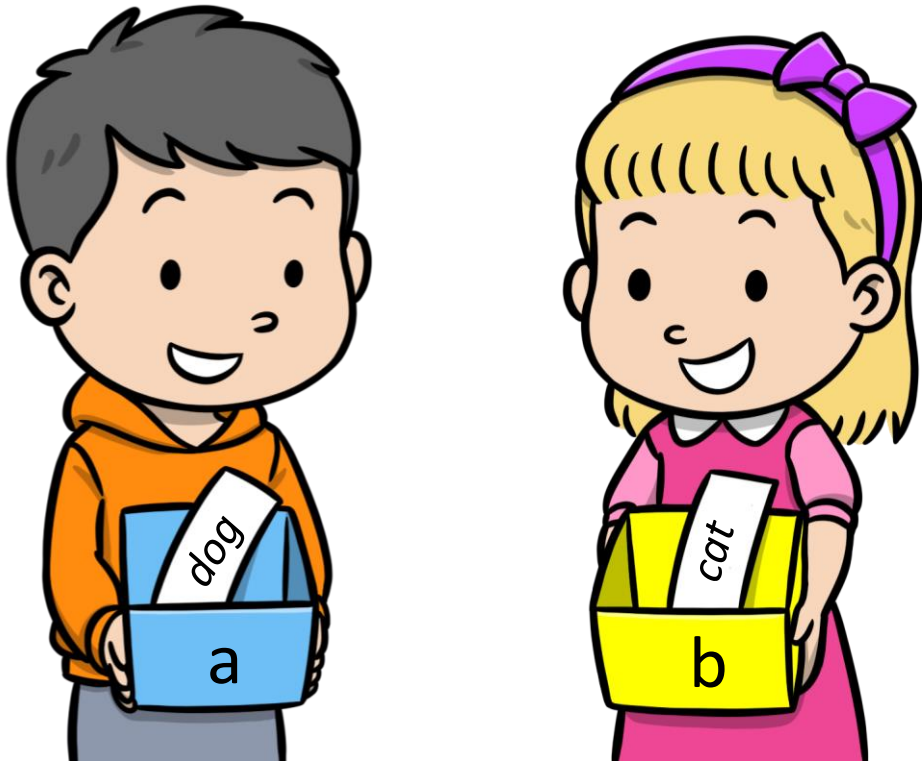
undefined



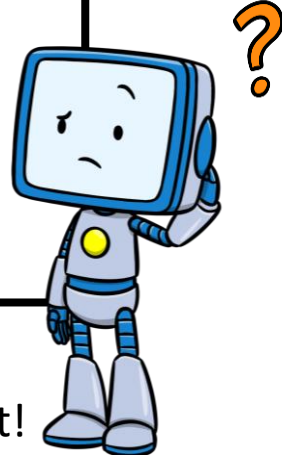
# Bonus tip: Exchanging the values of two variables

Let's say you have two variables  $a$  and  $b$ , containing the values "dog" and "cat".

How can we exchange the content of these variables so that  $a = \text{"cat"}$  and  $b = \text{"dog"}$  ?



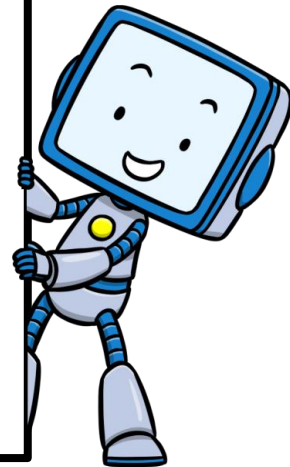
```
let a = "dog";  
let b = "cat";  
  
a = b;  
b = a;  
  
println(a);  
println(b);
```



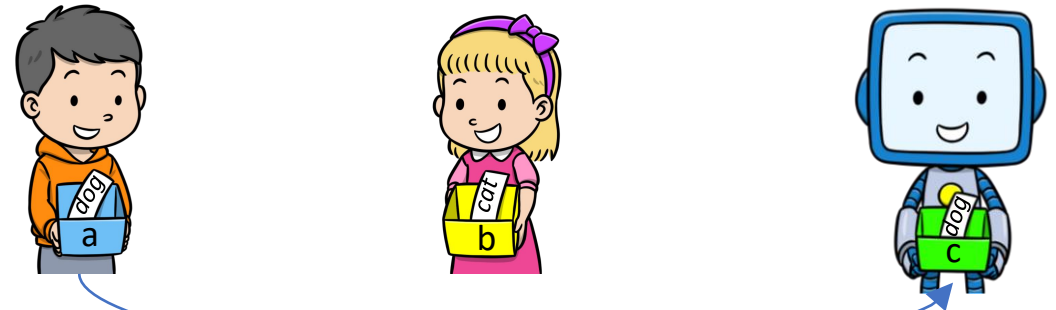
Naïve solution ... try to run the program to see the effect!

# Correct solution: Use a temporary 3<sup>rd</sup> variable

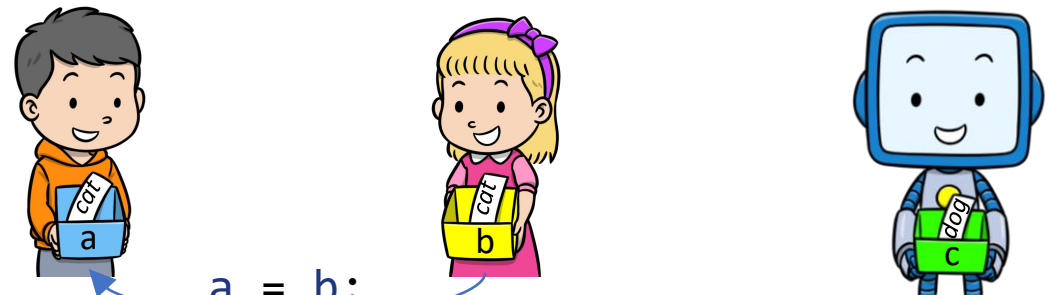
```
let a = "dog";  
let b = "cat";  
let c;  
  
c = a;  
a = b;  
b = c;  
  
println(a);  
println(b);
```



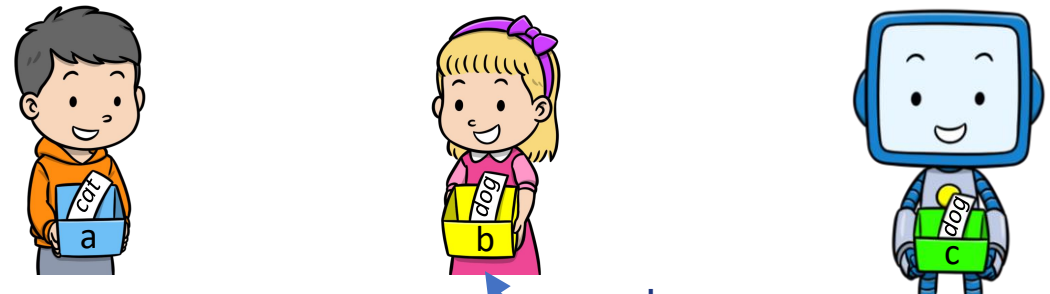
Run now the program! 



c = a;

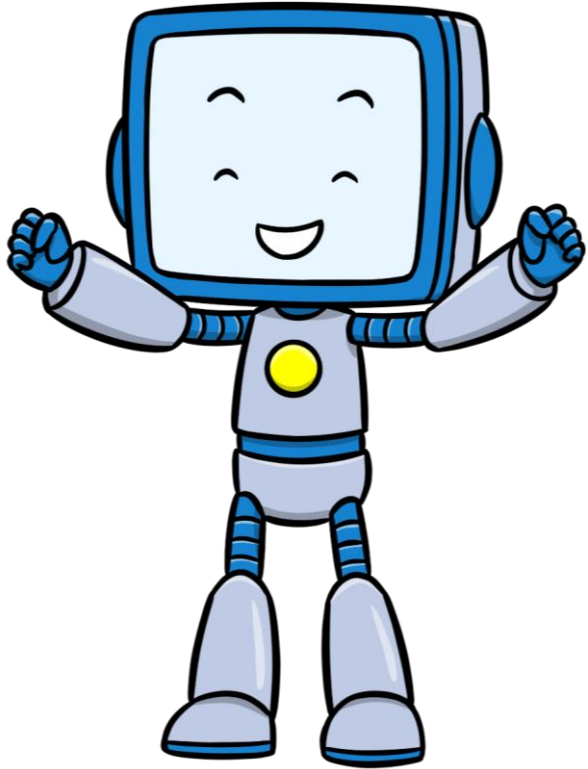


a = b;



b = c;

# Quick Recap



You can let the computer do calculations. At the end of the day, your computer is also a very powerful calculator!

```
circle(100, 100, 100);  
circle(100, 100, 96);
```



Program with numbers

```
circle(100, 100, 100);  
circle(100, 100, 100 - 4);
```



Program with expressions

```
let r2 = 100 - 4;  
circle(100, 100, 100);  
circle(100, 100, r2);
```



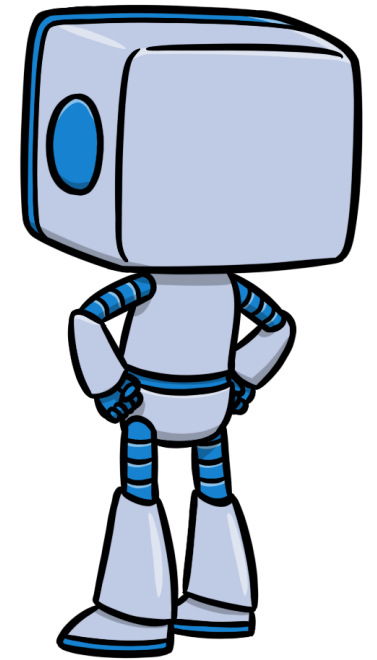
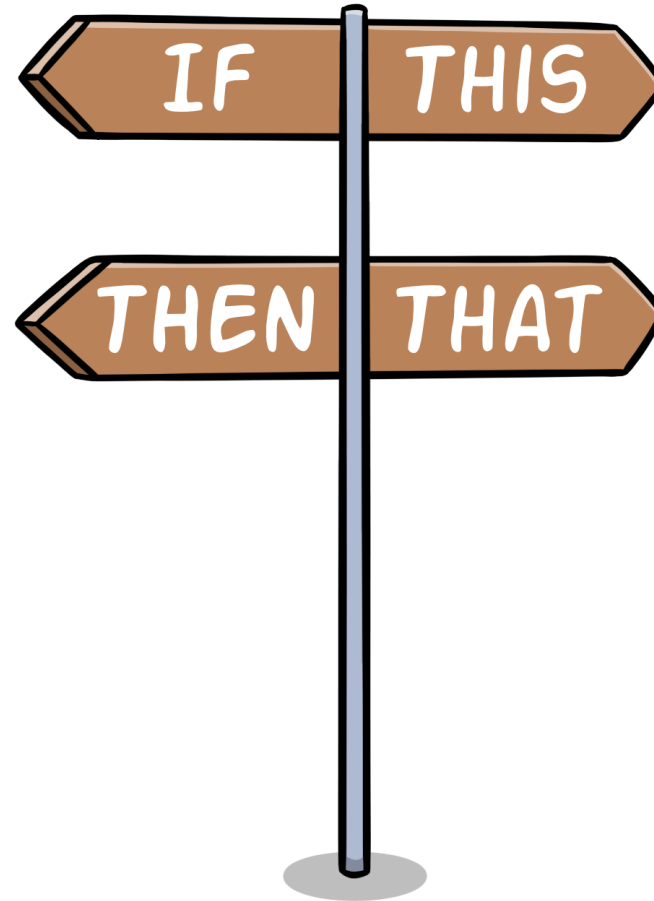
Program with variables



## Chapter VI – Conditional statement


- Deciding with *if*
- What about *else*?
- Cascading else-if statements
- Comparison operators
- **Exercise: Rating system**
- Boolean variables and logical expressions
- **Exercise: Solving the quadratic equation**
- Scope of variables

Let's teach the computer  
to take decisions...



# Introducing *if* statement

```
if ( condition )  
{  
    instruction 1  
    instruction 2  
    ...  
}
```



```
if (a > 0)  
{  
    println("a is ", a);  
    println("positive");  
}
```

Don't type this yet. Just analyze the syntax.

*if* statement makes possible to execute a block of instructions (aka *code block*) only if a certain condition is valid.

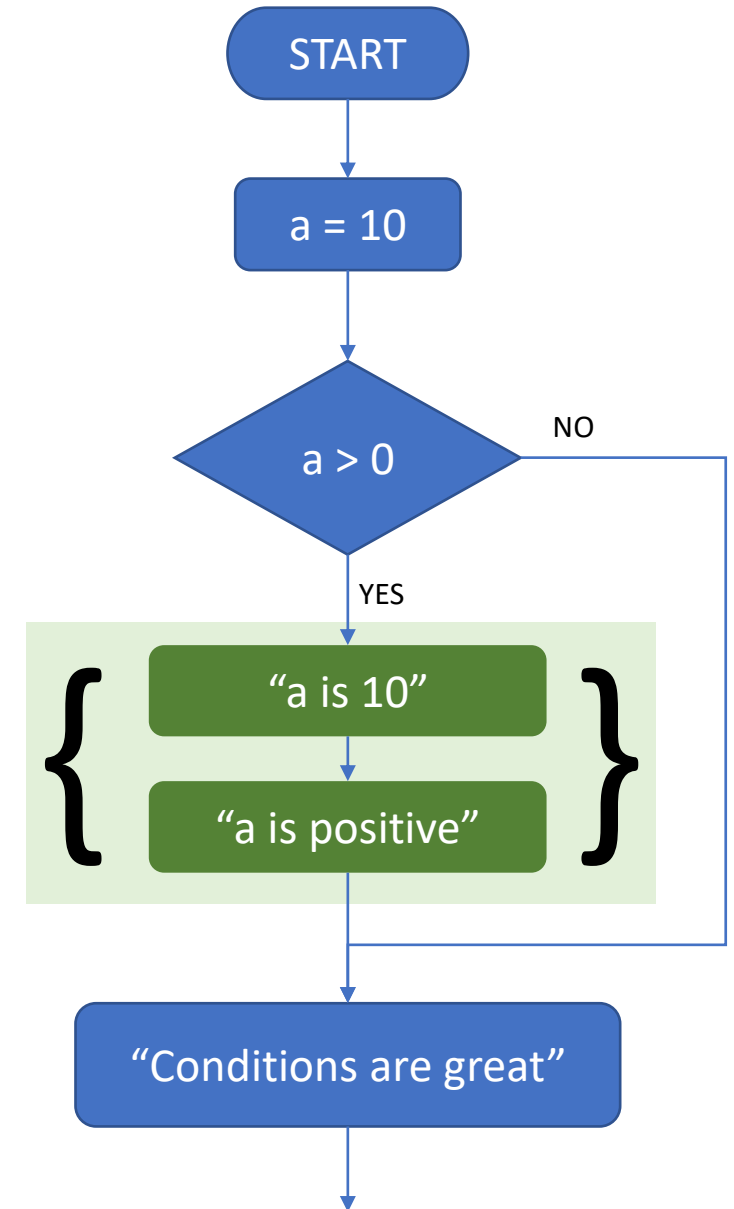
If the condition is not valid, the instructions between curly braces are not executed.

# Deciding with *if*...

```
let a = 10;  
  
if (a > 0)  
{  
    println("a is ", a);  
    println("a is positive");  
}  
  
println("Conditions are great");
```

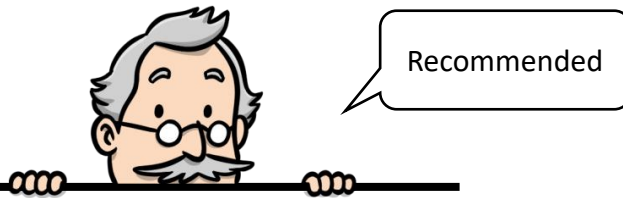


- Type carefully this small program and then run it. What is the output?
- Now modify the first line of code, and instead of 10 put there a negative number. What do you see now?



# About code blocks

- Remember to write readable code.
- Always indent the code inside the curly braces with one Tab key press `⌘TAB`. This is important especially if you have other statements with code blocks inside a code block (e.g. another *if* inside an *if*)
- JavaScript is flexible with placement of curly braces, but **we recommend** (especially for code newbies) to place curly braces one under the other so you can clearly see the code-block.



```
if (a > 0)
{
  ⌘TAB println("a is ", a);
  ⌘TAB println("a is positive");
}
```

```
if (a > 0) {
  println("a is ", a);
  println("a is positive");
}
```



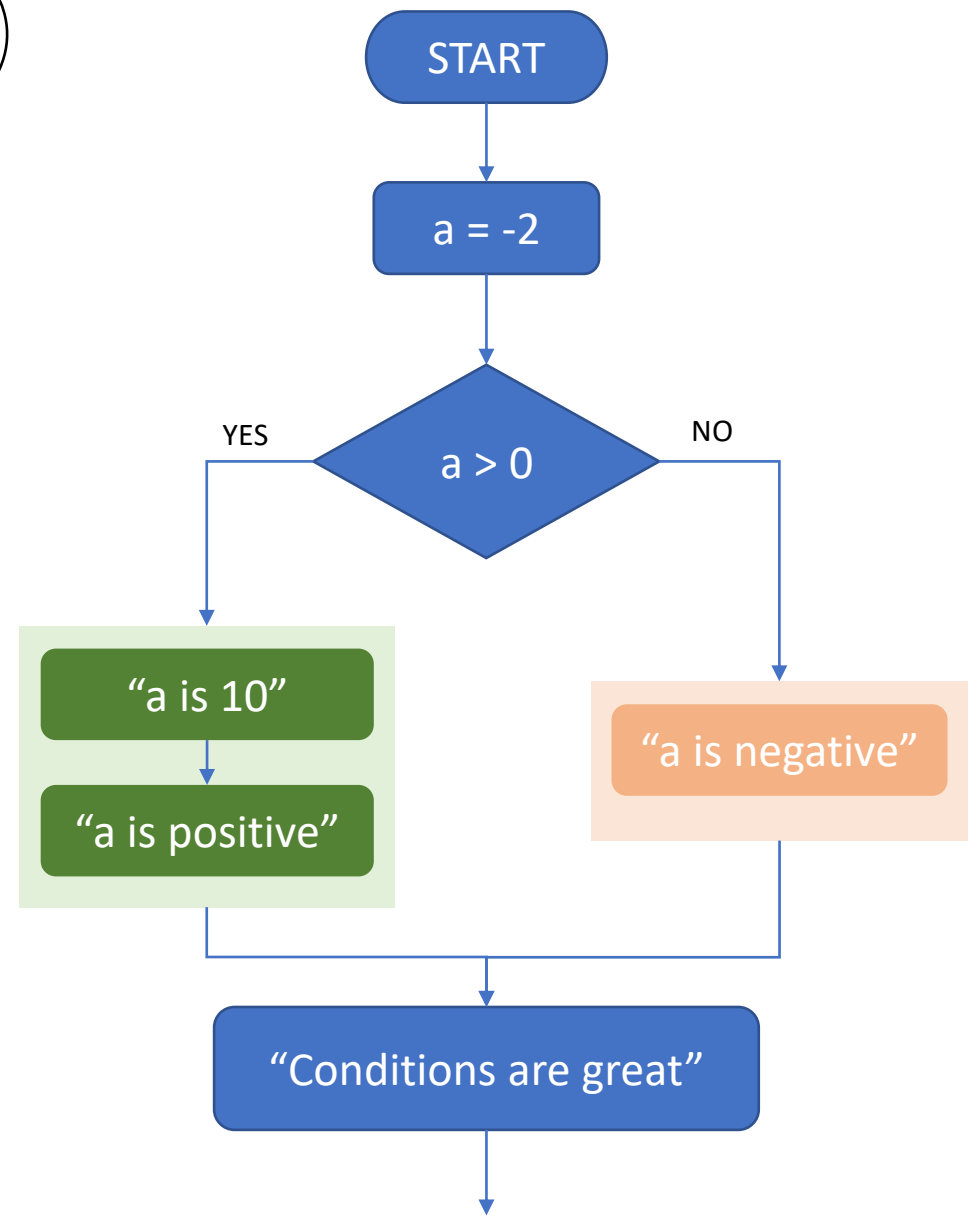
# What *else*?

```
let a = -2;  
  
if (a > 0)  
{  
  println("a is ", a);  
  println("a is positive");  
}  
  
else  
{  
  println("a is negative");  
}  
  
println("Conditions are great");
```

else block is executed if the *if* one is not



- Modify the program to include also an *else branch* followed by a new code block
- Don't use any parenthesis or symbol after else keyword!



# Cascading conditions with *else if* ...

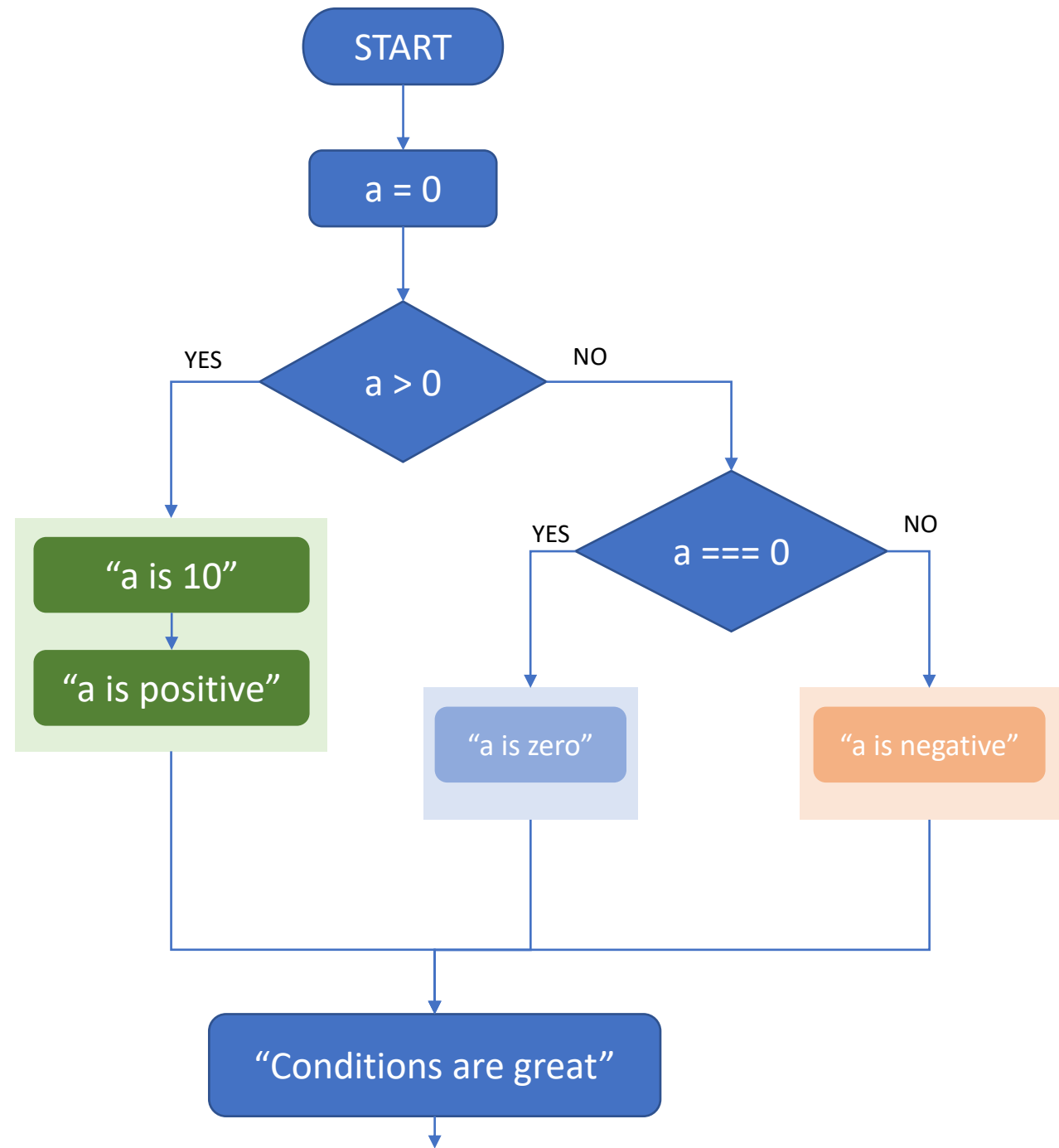
```
let a = 0;

if (a > 0)
{
  println("a is ", a);
  println("a is positive");
}

else if (a === 0)
{
  println("a is zero!");
}

else
{
  println("a is negative");
}

println("Conditions are great");
```



- Let's modify the code and add one more "branch" for an else if statement

## Independent *if* statements

```
let a = 10;

if (a > 10)
{
    println("a > 10");
}

if (a > 2)
{
    println("a > 2");
}

if (a > 5)
{
    println("a > 5");
}

if (a > 9)
{
    println("a > 9");
}
```



## Cascading else if statements

```
let a = 10;

if (a > 10)
{
    println("a > 10");
}
else if (a > 2)
{
    println("a > 2");
}
else if (a > 5)
{
    println("a > 5");
}
else if (a > 9)
{
    println("a > 9");
}
```



Type in the program on the left. It contains a series of unrelated *if* statements.



Run it and observe that all *if* statements seems to be evaluated.

Next, modify the program to add an `else` keyword in front of the indicated *ifs*. Now we have a big *if* / *else-if* statement with different branches.



Run it and observe that *if* statements are evaluated until the first one is found to match. Then the rest are skipped.

```
let a = 10;

if (a > 10)
{
    println("a > 10");
}

else if (a > 2)
{
    println("a > 2");
}

else if (a > 5)
{
    println("a > 5");
}

else if (a > 9)
{
    println("a > 9");
}
```



10

2

5

9

```
let a = 10;

if (a > 10)
{
    println("a > 10");
}

else if (a > 9)
{
    println("a > 9");
}

else if (a > 5)
{
    println("a > 5");
}

else if (a > 2)
{
    println("a > 2");
}
```



10

9

5

2

## Order of else-if blocks matter!

Always put the most specific condition first.

In the previous program, we switched around the order of else-if blocks. Notice now that the result is different.

# Operators



Besides arithmetic operators, JavaScript has also comparison operators that can be used inside *if* statements

## Arithmetic operators

Do you remember the arithmetic operators from the variables lesson?

+ - \* / ( )

$$100 / 2 = 50$$

$$(123 * 453) / 12 = 4643.25$$

$$3 + 10 / 2 = 8$$

## Comparison operators

A comparison expression is formed using comparison operators

>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to
===	Equal  (notice that we use === and no = to compare for equality. Single = is reserved for variable assignments)
!==	Different than

# Exercise: Rating system

Let's build a simple rating system using *if / else-if* statements.

The program needs to display the appropriate message based on the actual rating from variable *rating*

```
let rating = 5;
```

```
...  
println("Excellent!!!");
```

 ← If rating is 5!

```
...  
println("Good");
```

 ← If rating >= 4

```
...  
println("Average");
```

 ← If rating >= 3

```
...  
println("Below average");
```

 ← Otherwise

Excellent!!!



```
let rating = 5;

if (rating === 5)
{
    println("Excellent!!!");
}

else if (rating >= 4)
{
    println("Good");
}

else if (rating >= 3)
{
    println("Average");
}

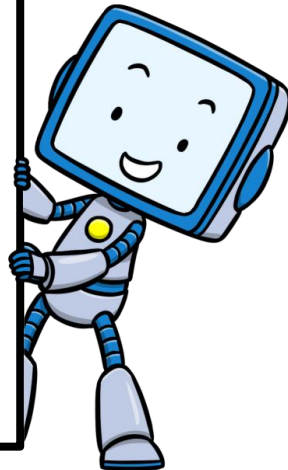
else
{
    println("Below average");
}
```



## Solution for: Rating system

- The program uses an if / else-if statement with cascading conditions
- Notice that order of conditions matter for our program (if you put conditions in a different order, you may get an incorrect result)

Compare this program with your version.



# Boolean variables

- JavaScript evaluates all conditions and logical expressions (also known as Boolean expressions) to either **true** or **false**.
- **true** and **false** are keywords inside JavaScript language
- Variables can be also of type Boolean – which means they hold a value that is either true or false, like the variable *isGoodRating*.

```
let rating = 4.5;  
let hasGoodRating;
```

```
if (rating > 4)  
{  
    println("Good rating!");  
    hasGoodRating = true;  
}
```

```
// Compare boolean variable with true  
if (hasGoodRating === true)  
{  
    println("Good rating!");  
}
```

```
// No need to put === true  
// when comparing to true  
if (hasGoodRating)  
{  
    println("Good rating!");  
}
```



# Logical expressions inside *ifs*

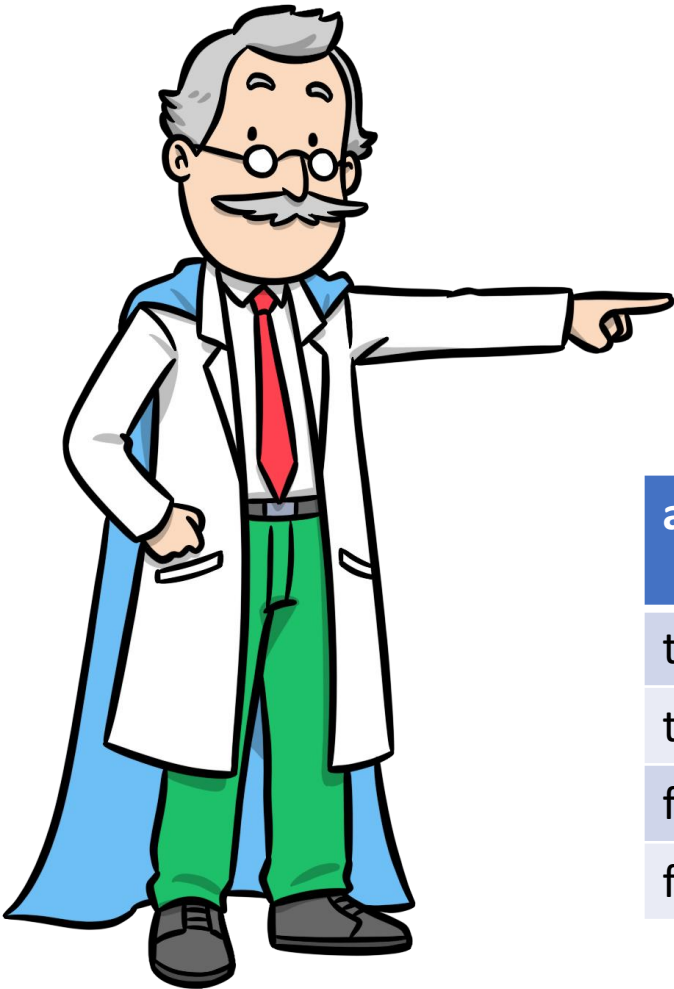
JavaScript has logical operators which enables to combine simple comparison expression in a bigger logical expression

JavaScript Operator	Meaning	JavaScript Example	Pseudo code
&&	AND	<code>if (a &lt; 10 &amp;&amp; b &gt; 5)</code>	IF a < 10 AND b > 5 THEN { ... }
	OR	<code>if (a &gt; 100    b &gt; 100)</code>	IF a > 100 OR b > 100 THEN { ... }
!	NOT	<code>if (!(a &lt; 10))</code>	IF NOT (a < 10) THEN { ... }



You can use parenthesis if you want to build even bigger logical expressions with multiple conditions.

# Truth tables



- Although you can intuitively tell the answer of a logical / Boolean expression, the following tables may help.
- They contain every combination possible for two value that may participate in a Boolean operation.

Value	NOT Value ! Value
true	false
false	true

a	b	a AND b a && b
true	true	true
true	false	false
false	true	false
false	false	false

a	b	a OR b a    b
true	true	true
true	false	true
false	true	true
false	false	false

# Example

If we know the x and y coordinates of a point, we can determine if it is inside the canvas by using a logical expression to check if they are in the range 0...800 and 0...600



```
let x = 400;
let y = 300;

if ( x >= 0 && x < 800 && y >= 0 && y < 600 )
{
    println("Inside canvas");
    circle(x, y, 10);
}

else
{
    println("Outside canvas");
}
```



Run this program and vary the x and y value to test both blocks.

# Example (cont.)

```
let x = 400;
let y = 300;

let insideCanvas = (x >= 0 && x < 800 && y >= 0 && y < 600);
let outsideCanvas = !insideCanvas;

if (insideCanvas)
{
    println("Inside canvas");
    circle(x, y, 10);
}

if (outsideCanvas)
{
    println("Outside canvas");
}
```



We modified the program, assigning the result of the logical expression to a variable named `insideCanvas`.

We also created its opposite `outsideCanvas` by using the NOT ! operator.

This technique enables us to check multiple times the variables without writing the full expression.

# Exercise: Solving the quadratic equation

(second-degree polynomial equation)

Quadratic equation is an equation that can be arranged as:

$$ax^2 + bx + c = 0 \quad a, b, c \text{ are the coefficients and are known numbers.}$$

A quadratic equation has two roots that can be found with formula:

$$x = \frac{-b \pm \sqrt{\Delta}}{2a} \quad \text{where } \Delta \text{ is the discriminant } \Delta = b^2 - 4ac$$

---

Before you solve the equation in JavaScript, try to **solve it on paper**.

What happens if  $a$  is zero, or  $\Delta$  is zero?

Now let's start coding!

```
const a = 2;
const b = 3;
const c = -5;

...

let x1 = ...
let x2 = ...

println("x1=", x1);
println("x2=", x2);
```

```
x1=1
x2=-2.5
```

# Step 1: Let's first check if $a$ is zero

```
const a = 0;
const b = 2;
const c = -5;

if (a === 0)
{
  println("First degree equation!");

  // to do: check also if b is zero

  let x = -c / b;
  println("x=", x);
}

else
{
}
```



When solving big problems, it is useful to work step by step, and test the program after each step.

If  $a === 0$ , then we have a first-degree equation.

$$x = -\frac{c}{b}$$



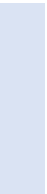
Type-in and run this program. Only after you tested carefully the first *if* branch, proceed to write the code for the *else* branch.

```
const a = 2;
const b = 3;
const c = -5;

if (a === 0)
{
  println("First degree equation!");


  // to do: check also if b is zero

  let x = -c / b;
  println("x=", x);
}

else
{
  
}
}
```

## Step 2: Add the *else* branch

Here we are coding the *happy-path* of the else branch, where delta is positive, and the equation has two real solutions.

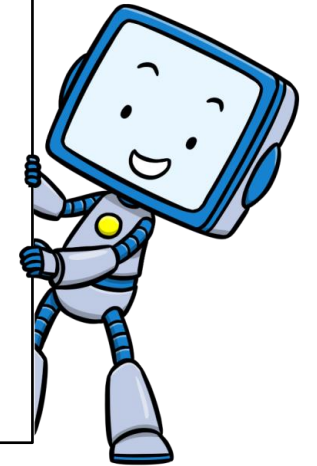
Try to run the program after this addition. Vary the parameters a, b and c and check if you encounter a situation where delta is negative. 

```
let delta = b * b - 4 * a * c;

let sqrtDelta = sqrt(delta);

let x1 = ( -b + sqrtDelta ) / ( 2 * a );
let x2 = ( -b - sqrtDelta ) / ( 2 * a );

println("x1=", x1);
println("x2=", x2);
```



New: To calculate square root we are using a mathematical function called `sqrt()`.

# Step 3: Enhance!

```
const a = 2;
const b = 3;
const c = -5;

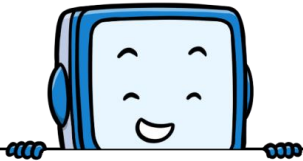
if (a === 0)
{
    println("First degree equation!");

    // to do: check also if b is zero

    let x = -c / b;
    println("x=", x);
}

else
{
    ←
}

}
```



Update the code on the *else* branch as you see on the right. We are now testing for different values of delta.

```
let delta = b * b - 4 * a * c;

if (delta > 0)
{
    let sqrtDelta = sqrt(delta);

    let x1 = ( -b + sqrtDelta ) / (2 * a);
    let x2 = ( -b - sqrtDelta ) / (2 * a);

    println("Equation has 2 real roots!");
    println("x1=", x1);
    println("x2=", x2);
}

else if (delta === 0)
{
    println("Equation has 1 root!");
    println("x1 = x2 = ", -b / (2 * a) );
}

else
{
    println("Equation has 2 complex roots!");
}

}
```



# Step 3: Complete listing

- Our quadratic equation solver program is done! It contains a big number of lines ... therefore it barely fits on the screen.
- Don't worry – if you built the program step by step, as presented before, you don't have to type it again.
- Please note the nested if statements as well as the code indentation inside the blocks to help with reading of the code.

We left unimplemented a small check for the case when  $a = 0$  and  $b = 0$ . You can solve this as a homework.



```
const a = 2;
const b = 3;
const c = -5;

if (a === 0)
{
    println("First degree equation!");

    // to do: check also if b is zero

    let x = -c / b;
    println("x=", x);
}

else
{
    let delta = b * b - 4 * a * c;

    if (delta > 0)
    {
        let sqrtDelta = sqrt(delta);

        let x1 = ( -b + sqrtDelta ) / (2 * a);
        let x2 = ( -b - sqrtDelta ) / (2 * a);

        println("Equation has 2 real roots!");
        println("x1=", x1);
        println("x2=", x2);
    }

    else if (delta === 0)
    {
        println("Equation has 1 root!");
        println("x1 = x2 = ", -b / (2 * a) );
    }

    else
    {
        println("Equation has 2 complex roots!");
    }
}
}
```

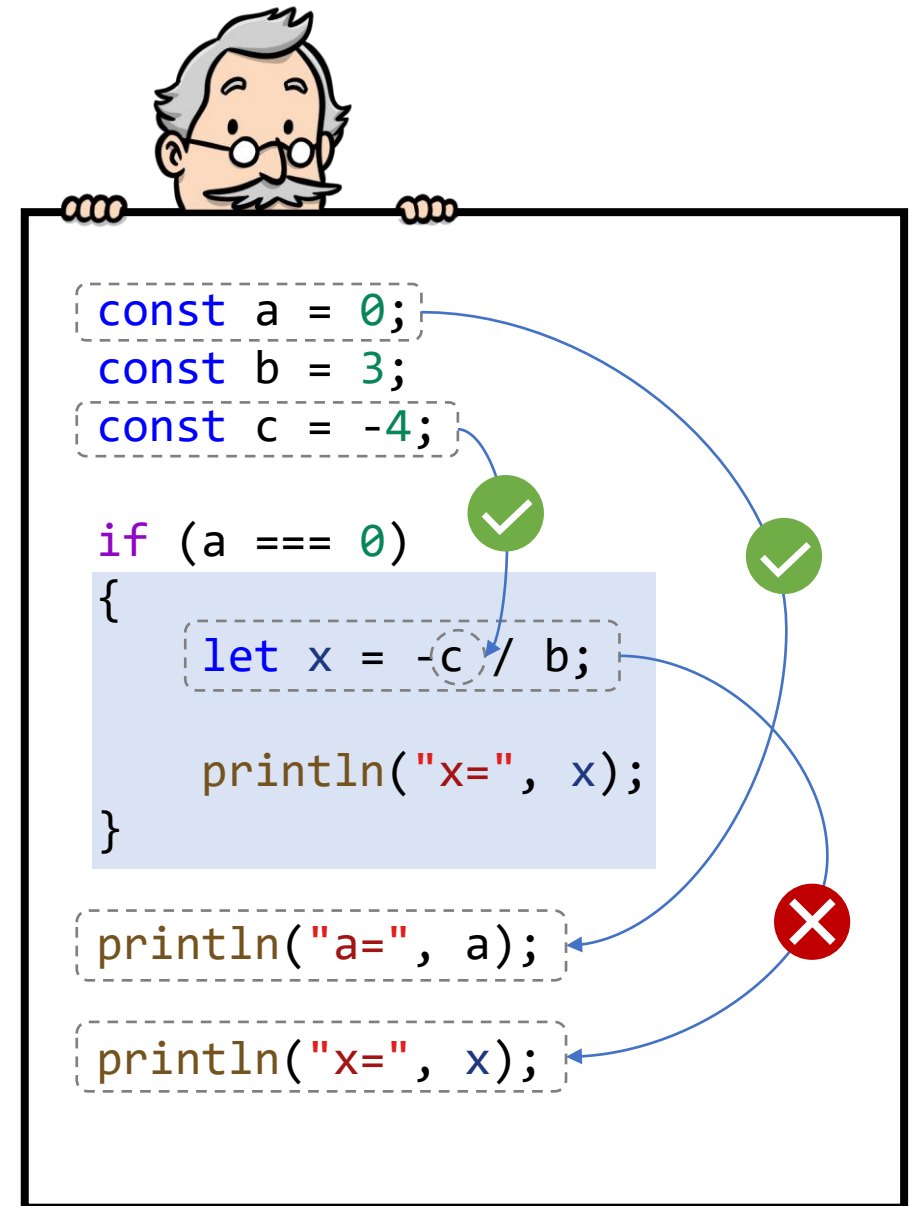


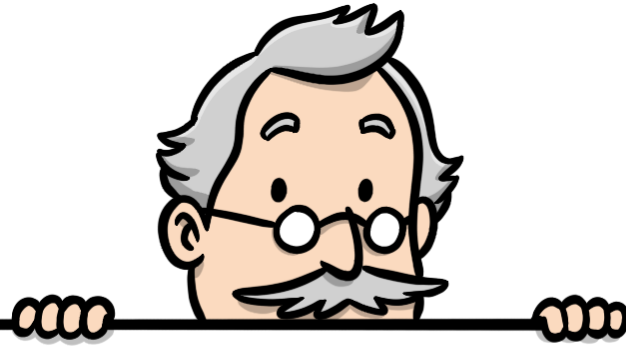
# Scope of variables

- Variables declared outside any code block are considered “*global variables*”. They are visible in all other code blocks and through the entire program.
- Variables defined inside a code block are considered “*local variables*”. They are visible only in that code block. You can declare variables with the same name in different code blocks. They don’t interfere with each other since code blocks are small boxes that hold variables inside.



Try this: In the previous program, add a `println` line to print a block variable outside the block.





## Chapter VII – for loops

- How can we repeat an instruction several times?
- *for* loop syntax
- Common uses of the *for* loop
- **Exercise:** Sum of numbers from 1 to 10
- **Exercise:** Factorial of n
- **Exercise:** Display multiplication table
- **Exercise:** Draw concentric circles
- **Exercise:** Lines with *for*
- **Exercise:** Draw multiple cars
- **Exercise:** Color shades
- **Exercise:** Graph sine function

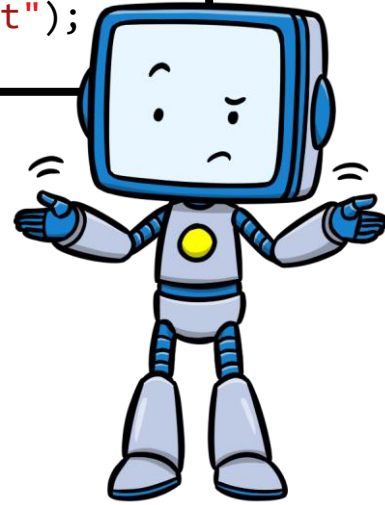
# Repeat the message



I like to code in JavaScript  
I like to code in JavaScript  
I like to code in JavaScript  
I like to code in JavaScript  
I like to code in JavaScript  
I like to code in JavaScript  
I like to code in JavaScript  
I like to code in JavaScript  
I like to code in JavaScript  
I like to code in JavaScript  
I like to code in JavaScript  
I like to code in JavaScript

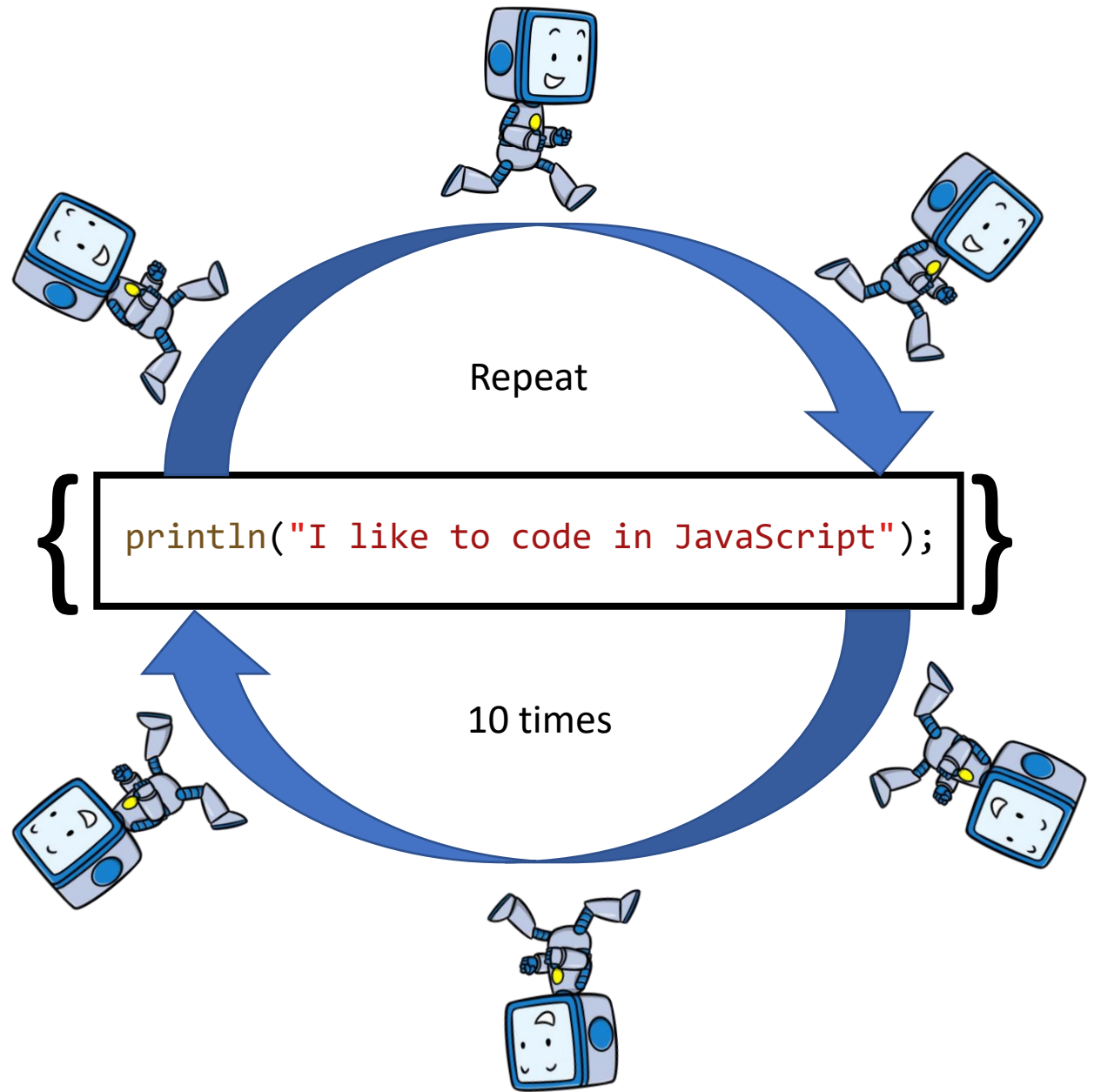


```
println("I like to code in JavaScript");  
println("I like to code in JavaScript");  
println("I like to code in JavaScript");  
println("I like to code in JavaScript");  
println("I like to code in JavaScript");  
println("I like to code in JavaScript");  
println("I like to code in JavaScript");  
println("I like to code in JavaScript");  
println("I like to code in JavaScript");  
println("I like to code in JavaScript");  
println("I like to code in JavaScript");  
println("I like to code in JavaScript");
```



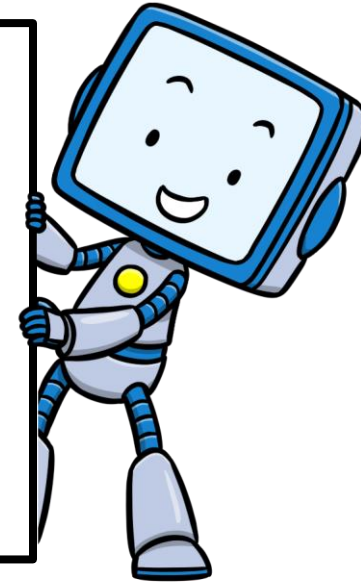
Can we ask the computer to repeat this line for me?

Let's ask the computer to repeat this line...



# Repeating a code block 10 times

```
for(let i = 0; i < 10; i++)  
{  
  Your lines of code  
}
```



- Each time you need to repeat a line of code (or multiple lines) several times, use this for loop template.... Please write it in your notebook.
- You only need to add your lines of code inside the curly braces and specify how many times you need to repeat.

Let's try it!

```
for(let i = 0; i < 10; i++)  
{  
    println("JavaScript");  
}
```

Copy the above code in a new program. Make sure you include every symbol as you see on the screen.



**Step 1:** Write the following snippet of code exactly as you see on the screen.

```
for(let i = 0; i < 10; i++)  
{  
}
```

**Step 2:** The only part you may want to modify, at this time, is number 10. This specifies how many times to execute the instructions inside { }

**Step 3:** Put the instruction(s) you want to repeat in the for loop *code block*, e.g. inside { }

# Reading the for loop the easy way...

FOR

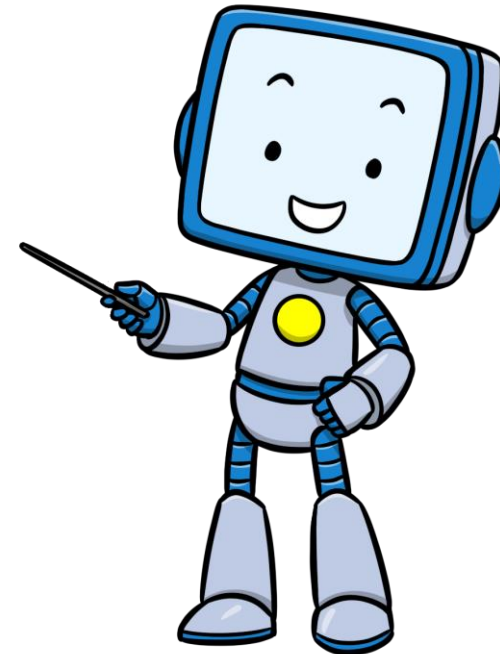
`i = 0`

TO

`10 (exclusive)`

Execute block { }

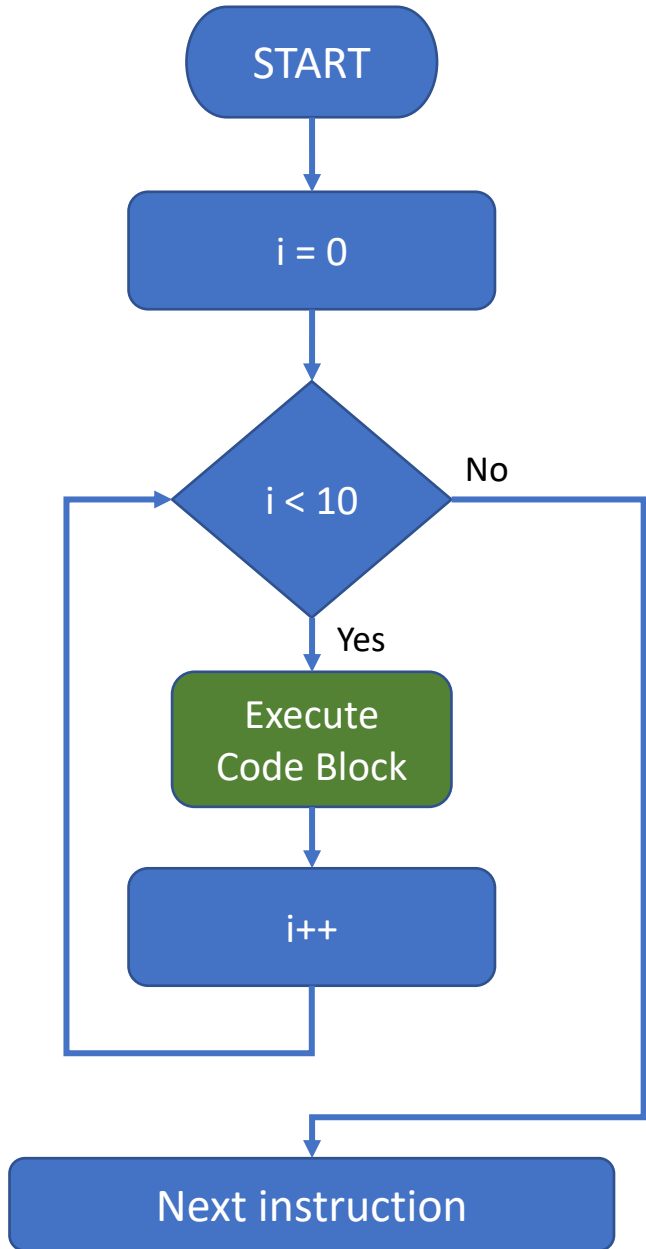
```
for(let i = 0; i < 10; i++)  
{  
  println("JavaScript");  
}
```







# *for* works by utilizing a variable...



← Step 1: JavaScript executes the initialization part just once for each for loop. Here we're declaring variable *i*.

← Step 2: Then checks the condition, and if the condition is satisfied, it will execute the code block, otherwise the for loop is ending.

← Step 3: Code Block is executed

← Step 4: After code block execution the variable is updated according to the statement in the 3<sup>rd</sup> position. Here we're incrementing *i*.

↑ Step 5: JavaScript repeats from Step 2

# Accessing for loop variable inside the code block

- Did you know that you can make use of the for variable inside the code block?
- The code block is executed  $n$  times, and each time  $i$  has a different value:

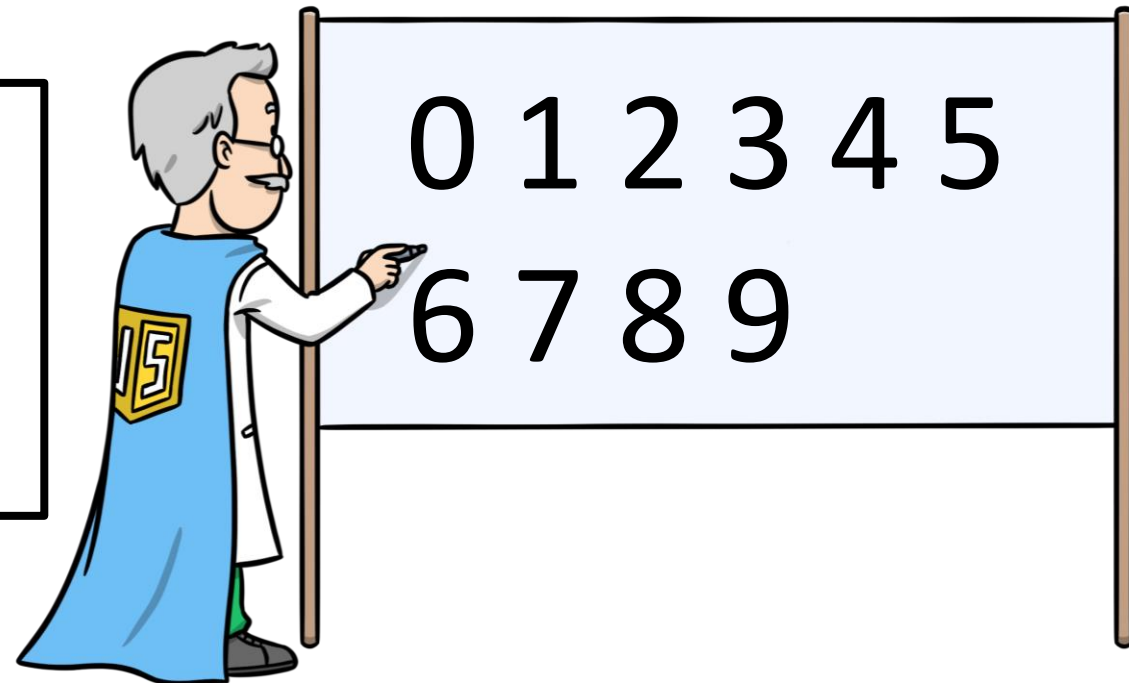
$i = 0$  then execute {...}

$i = 1$  then execute {...}

$i = 2$  then execute {...}

...

```
for(let i = 0; i < 10; i++)  
{  
    println(i);  
}
```



# Counting with *for*. Adjusting the lower and upper bounds

1

```
for(let i = 0; i < 10; i++)  
{  
    println(i);  
}
```

0 1 2 3 4 5 6 7 8 9

3

```
for(let i = 1; i < 11; i++)  
{  
    println(i);  
}
```

1 2 3 4 5 6 7 8 9 10

2

```
for(let i = 0; i < 10; i++)  
{  
    println(i + 1);  
}
```

1 2 3 4 5 6 7 8 9 10

4

```
for(let i = 1; i <= 10; i++)  
{  
    println(i);  
}
```

1 2 3 4 5 6 7 8 9 10

Can you spot the differences between these programs? How can you explain their output?

# Counting down

If in the 3<sup>rd</sup> section of for, you select to decrement the variable (rather than incrementing), you can count down.

## Counting by 1 (same code as before)

```
for(let i = 0; i < 10; i++)  
{  
    println(i);  
}
```

0 1 2 3 4 5 6 7 8 9

## Counting down by 1

```
for(let i = 10; i > 0; i--)  
{  
    println(i);  
}
```

10 9 8 7 6 5 4 3 2 1

Feel free to update the lower and upper bound in this count-down template.



# Counting up and down with a different step

Sometimes is needed to use a counting step different than 1... In the following examples you can see counting up and down with various steps.

## Counting by 2 (display even numbers)

```
for(let i = 0; i < 10; i += 2)
{
    println(i);
}
```

0 2 4 6 8

## Counting down by 5

```
for(let i = 30; i >= 0; i -= 5)
{
    println(i);
}
```

30 25 20 15 10 5 0



Practice *for* loops by doing the following exercises:

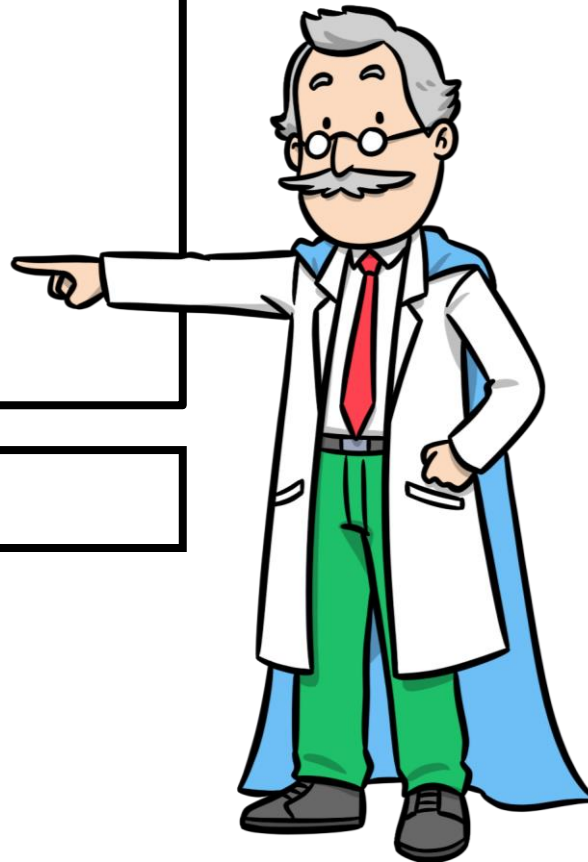
- Print all integer numbers between 10 and 20
- Print, in inverse order, all integer numbers between 20 and 10 (count down)
- Print all odd numbers between 1 and 29
- Print, in inverse order, all even numbers between 20 and 0

# You can break out of a *for loop*

```
for(let i = 0; i < 10; i++)  
{  
  println(i);  
  
  if (i >= 5)  
  {  
    break;  
  }  
}
```



0 1 2 3 4 5



`break;` is a useful keyword. You can use it to break out a for loop at any time.

Notice that `break` is not a typical command, therefore you don't invoke it with `()`.

We will see more uses of *for - break* in the future lessons.

# for Code Blocks

FOR i = 0 TO 10 REPEAT THESE INSTRUCTIONS

```
{  
    println("I");  
    println("like");  
    println("JavaScript");  
}
```

The *for* statement is followed by a code block (e.g. a group of instructions enclosed by curly braces { ... })

This allows us to repeat multiple instructions together for each step in the *for* loop.

As with the *if* code block, you can add multiple instructions in a code block





# Scope of variables



```
let s = "Hello";  
for(let i = 0; i < 10; i++)  
{  
  println(s);  
  println(i);  
}  
  
println(i); ❌
```

The code block of a *for loop* has the same rules as the code block we used for the *if* statement.

- Variables declared outside any code block are considered “*global variables*”. They are visible in all other code blocks and through the entire program.
- Variables defined inside a code block are considered “*local variables*”. They are visible only in that code block.
- Variables defined in the *for* line (e.g. *let i = 0;*) are visible in the code block of that particular *for* only.

*for loops are fun...*



... let's see some exercises

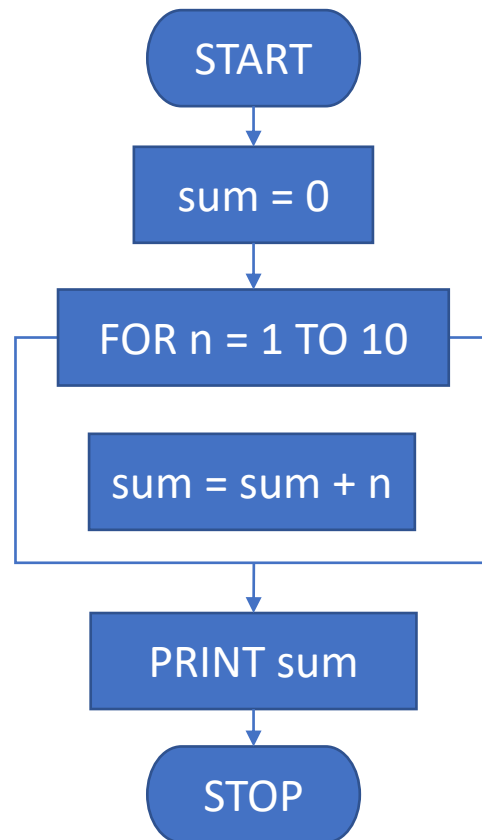
# Exercise: Sum of numbers from 1 to 10

- Let's use a *for* loop to calculate the sum of numbers from 1 to 10



We will use a simple algorithm that even us humans are using when we're calculating the sum of a series of numbers:

We add all the numbers, one by one, to the total.



Starting total (sum): 0

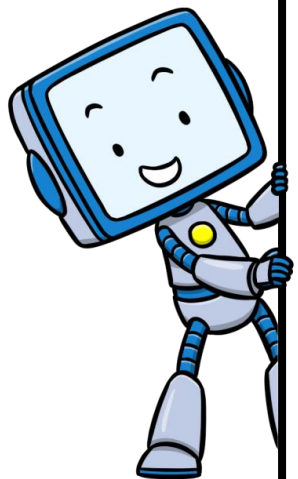
Old sum + Number = New sum

0	+	1	=	1
1	+	2	=	3
3	+	3	=	6
6	+	4	=	10
10	+	5	=	15
15	+	6	=	21
21	+	7	=	28
28	+	8	=	36
36	+	9	=	45
45	+	10	=	55



# Exercise: Sum of numbers from 1 to 10

- Let's use a *for* loop to calculate the sum of numbers from 1 to 10
- We will use a simple algorithm that even us humans are using when we're calculating the sum of a series of numbers: we add them one by one to the total.



```
let sum = 0;
for(let i = 1; i <= 10; i++)
{
    sum += i;
    // println(sum);
}
println("Sum=", sum);
```

Our total (e.g. variable *sum*) is initial 0

We will cycle through all the numbers from 1 to 10 using a *for* loop

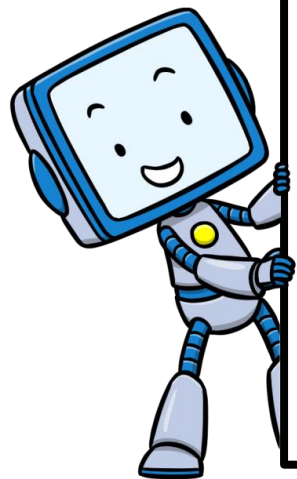
And add each number to variable *sum* (e.g. our total grows bigger with each number added)

We display the total *sum*

Tip: Uncomment the `println` line inside the for loop to inspect the *sum* variable as it grows!

# Exercise: Factorial of 10

- From math, we know that factorial of 10 is the product of all numbers from 1 to 10.  
 $10! = 1 * 2 * 3 * \dots * 10$
- We are using the same algorithm that we used for sum of numbers to calculate factorial. The only difference is that our total variable will be initiated with 1 (neutral element to multiplication)



```
let prod = 1;
for(let i = 1; i <= 10; i++)
{
    prod = prod * i;
}
println("Factorial=", prod);
```

Our total (e.g. variable *prod*) is initial 1

We will cycle through all the numbers from 1 to 10 using a *for* loop

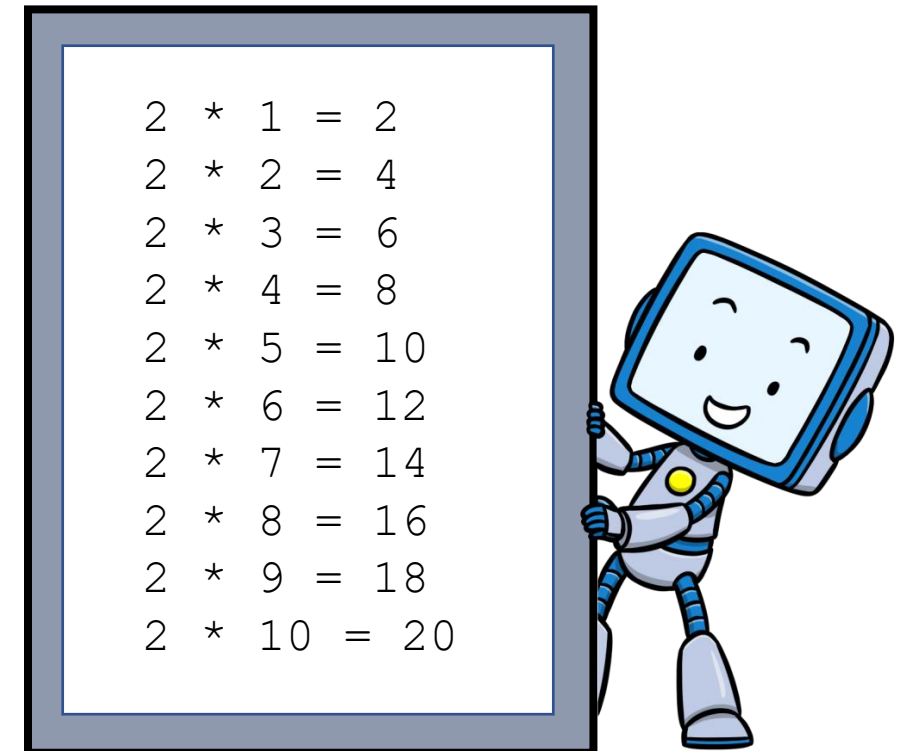
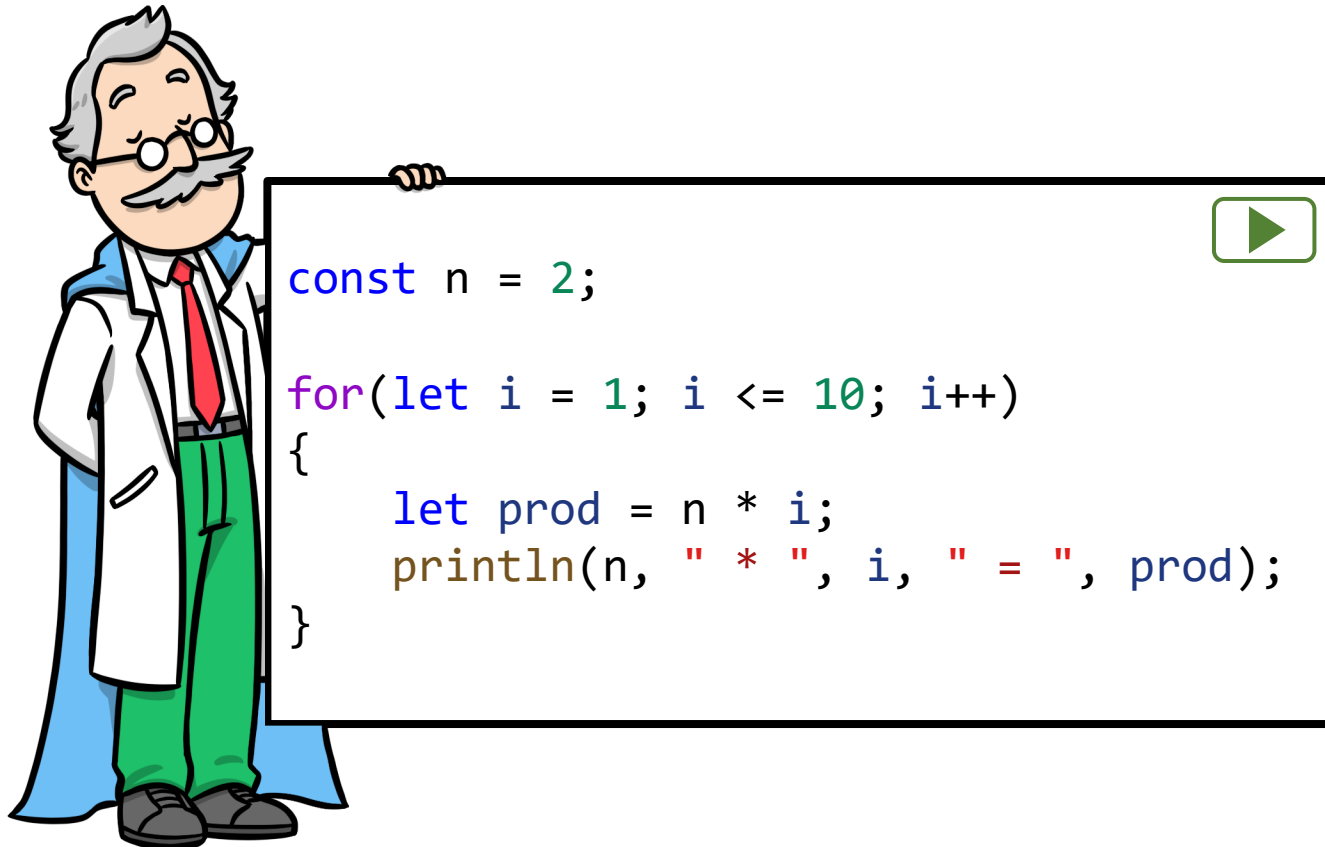
Multiply the previous product hold in *prod* variable to the new number

We display the total *prod*

Tip: Add a `println` line inside the for loop to inspect the variable *prod* as it grows!

# Exercise: Display multiplication table

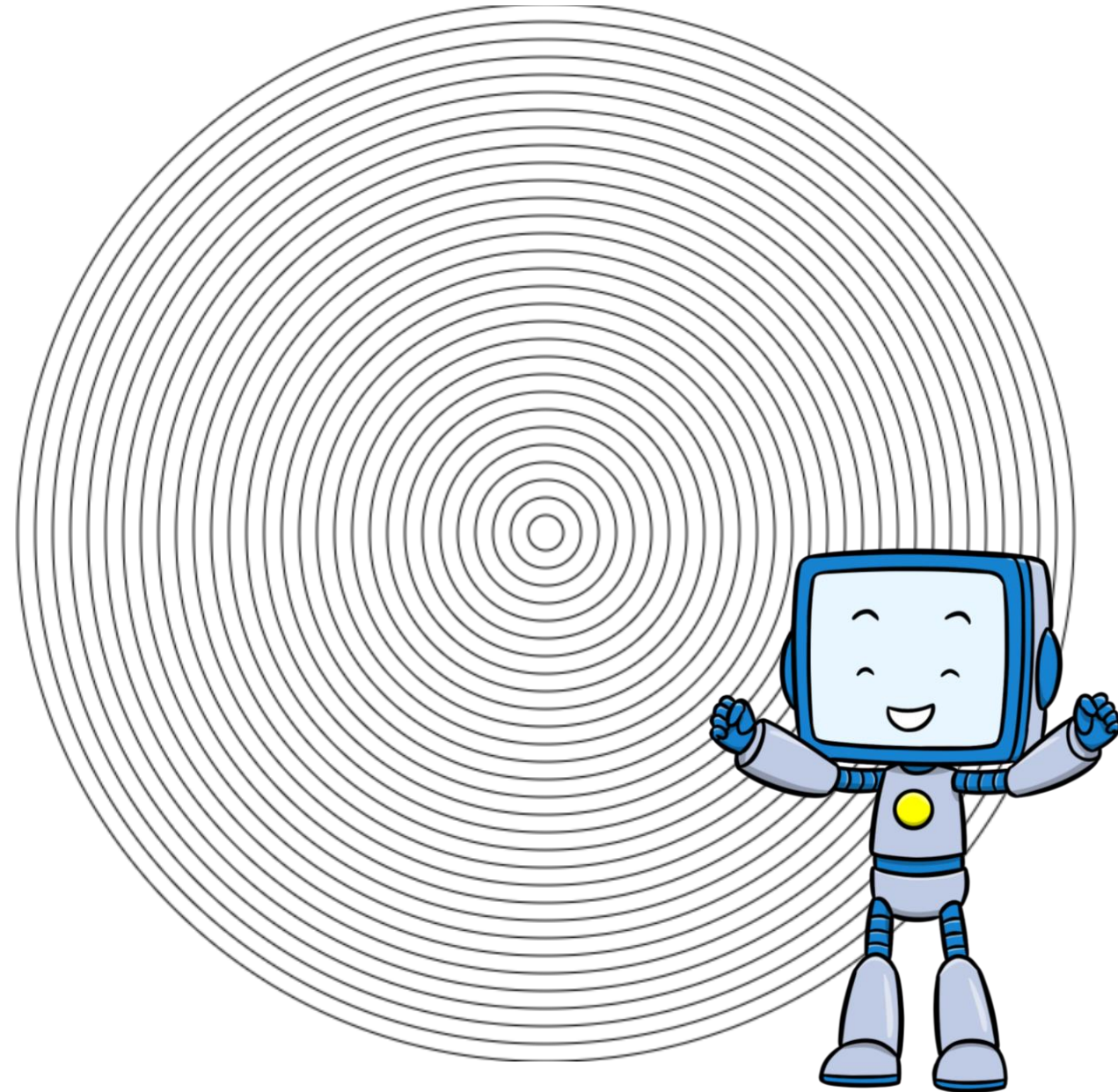
- Let's display the multiplication table with n (where n = 1, 2, ...)
- We don't have any *difficult* algorithm here, just a coordinated print of several pieces of text and numbers



# Exercise: Draw concentric circles

- Let's try now a graphical program with *for*.
- This enables us to practice *for* on a bigger range and with a different step to obtain these concentric circles.
- Play with the numbers to adjust the effect.

```
for(let r = 300; r >= 0; r -= 10)
{
  circle(400, 300, r);
}
```



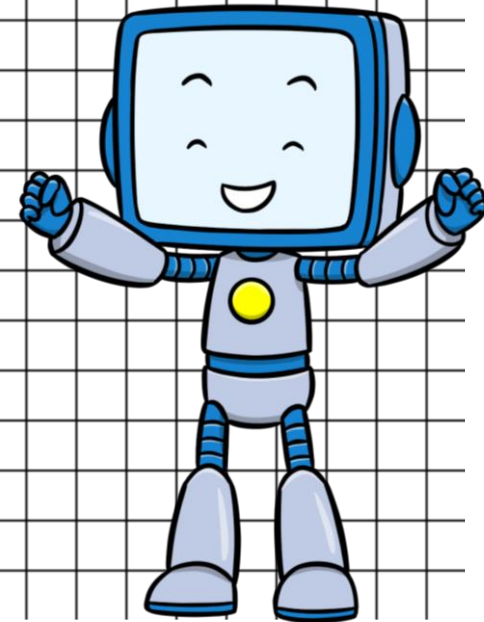
# Exercise: Horizontal and vertical lines

- Feel free to add as many *for* loops as you need in your program
- This one has two *for* loops to draw horizontal lines and then vertical lines

```
const squareSize = 25;

// Horizontal lines
for(let y = 0; y < 600; y += squareSize)
{
    line(0, y, 800, y);
}

// Vertical lines
for(let x = 0; x < 800; x += squareSize)
{
    line(x, 0, x, 600);
}
```

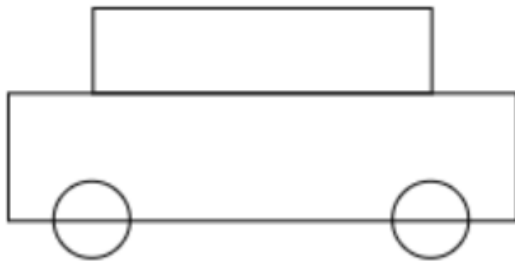




# Exercise: Row of cars

Do you remember this small program that we wrote in the variables lesson?

It draws a simple car on the screen.



Don't type it again. You should find it under "My Programs".

Please create a copy of it and update it with a *for* loop so will display a row of cars instead of a single car.

```
let x = 100;  
let y = 100;
```

```
const w = 200;  
const h = 50;  
const rp = 0.3;
```

```
// Calculate the width and height of top part  
let wt = (2 / 3) * w;  
let ht = (2 / 3) * h;
```

```
// Coordinates of top rectangle  
let side = (w - wt) / 2;  
let xt = x + side;  
let yt = y;
```

```
// Coordinates of bottom rectangle  
let xb = x;  
let yb = y + ht;
```

```
// Coordinates of wheels  
let xw1 = x + side;  
let xw2 = x + w - side;  
let yw = y + ht + h;  
let rw = h * rp;
```

```
rect(xt, yt, wt, ht);  
rect(xb, yb, w, h);  
circle(xw1, yw, rw);  
circle(xw2, yw, rw);
```

```
const y = 100;
const w = 60;
const h = 15;

for(let x = 0; x < 800; x += 80)
{
    // Calculate the width and height of top part
    let wt = (2 / 3) * w;
    let ht = (2 / 3) * h;

    // Coordinates of top rectangle
    let side = (w - wt) / 2;
    let xt = x + side;
    let yt = y;

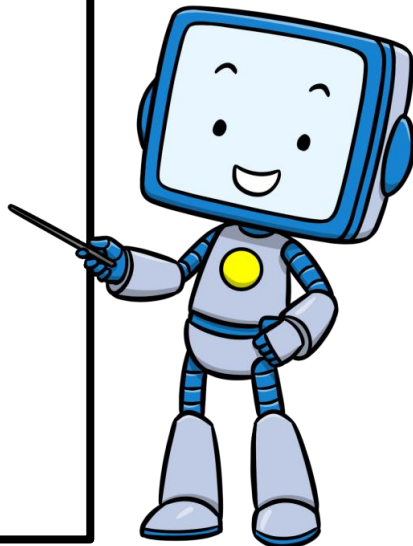
    // Coordinates of bottom rectangle
    let xb = x;
    let yb = y + ht;

    // Coordinates of wheels
    let xw1 = x + side;
    let xw2 = x + w - side;
    let yw = y + ht + h;
    let rw = h * 0.3;

    rect(xt, yt, wt, ht);
    rect(xb, yb, w, h);
    circle(xw1, yw, rw);
    circle(xw2, yw, rw);
}
```

## Exercise: Row of cars (cont)

- This is our version. We place the code from the previous program in code block of a for loop (we also slightly updated the car code to draw a smaller car).
- The for loop is cycling on x coordinates from 0 till end of screen (with a step of 80 – bigger than the car).
- y coordinate is constant and is defined at the beginning of the program



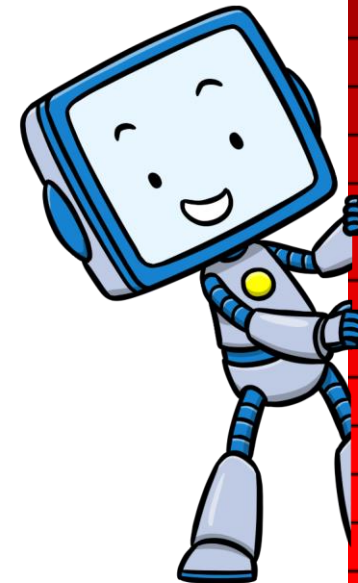
# Exercise: Color shades

- This small program draws a series of red shades.
- As you remember each color has three primary component: RED, GREEN and BLUE
- The program uses a *for* loop to cycle the RED component then draw a colored band using a filled rectangle.

```
const bandSize = 20;
const noBands = 600 / bandSize;

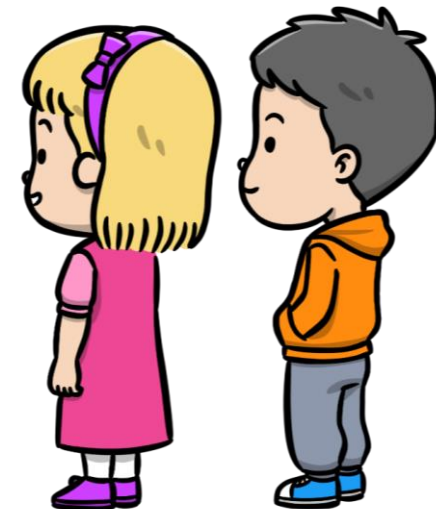
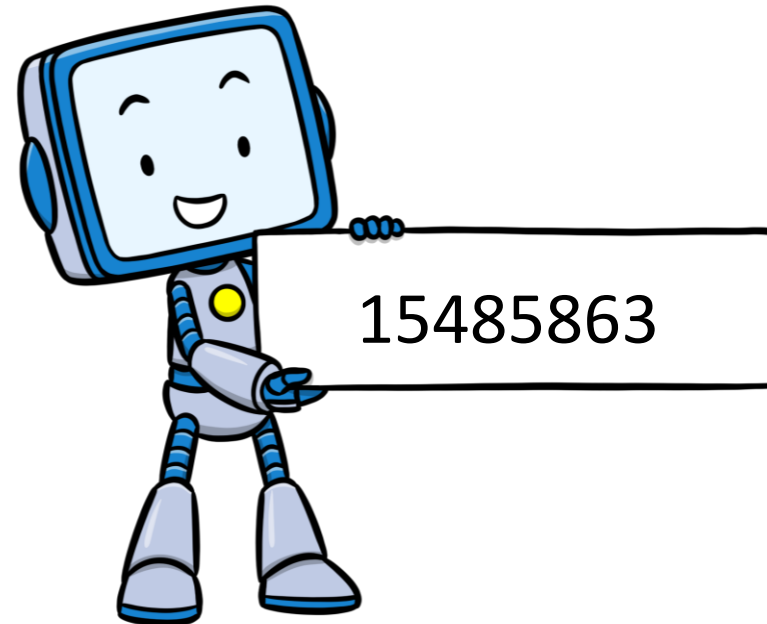
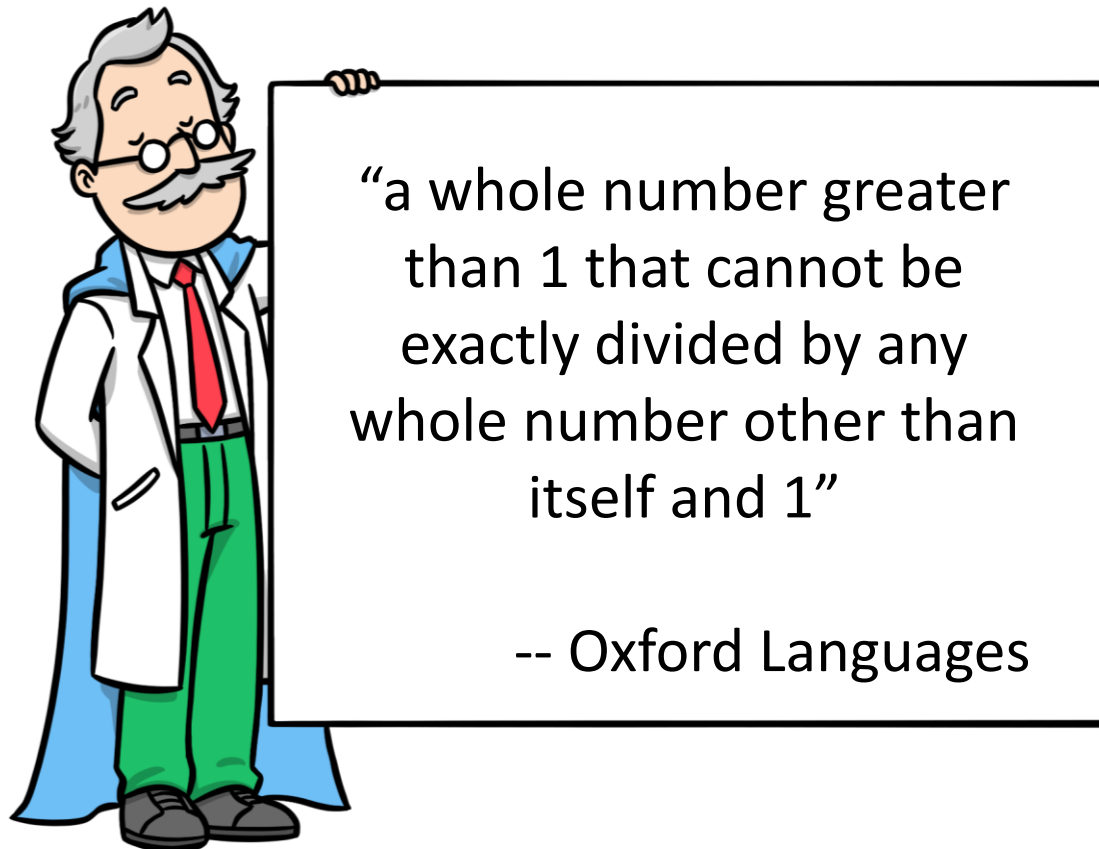
for(let i = 0; i < noBands; i++)
{
    let y = bandSize * i;
    let clr = i * 10;

    fill(clr, 0, 0);
    rect(0, y, 800, bandSize);
}
```



# Exercise: Finding prime numbers

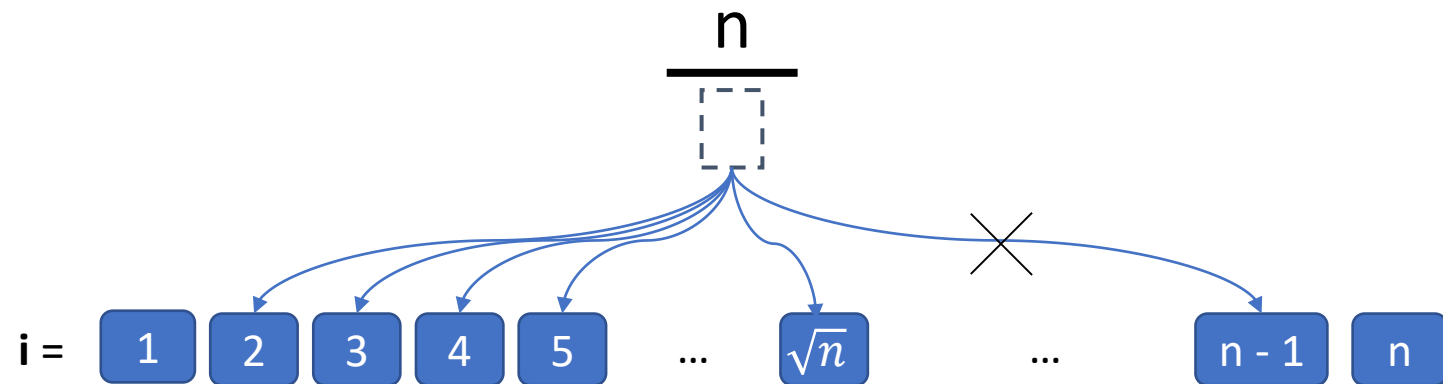
- Let's try a math exercise.
- Let's write a JavaScript program that will determine if a number is prime or not!



## Algorithm

According to the definition, to check if number  $n$  is prime, we can divide it with all numbers bigger than 1 and smaller than  $n$ . If cannot be exactly divided, then is prime!

Here we can do a small optimization: we can stop the check at  $\sqrt{n}$ . Why? There is a simple mathematical proof for this. We let this open for your discovery.



$n$  is the number we want to check if is prime

We assume  $n$  is prime

We loop through all the numbers up to  $\sqrt{n}$ .  
(in JavaScript we use `sqrt(n)` to calculate  $\sqrt{n}$ )

`%` is a special operator in JavaScript.  
It gives us the remainder of division.

If remainder is 0, then the two numbers can be exactly divided, and we conclude that our number is not prime. We break the for to stop checking against other numbers.

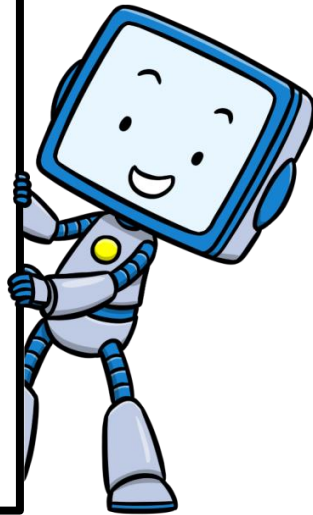
We print the result according to the Boolean variable `isPrime`.

```
const n = 17;
let isPrime = true;

for(let i = 2; i <= sqrt(n); i++)
{
    if (n % i === 0)
    {
        isPrime = false;
        break;
    }
}

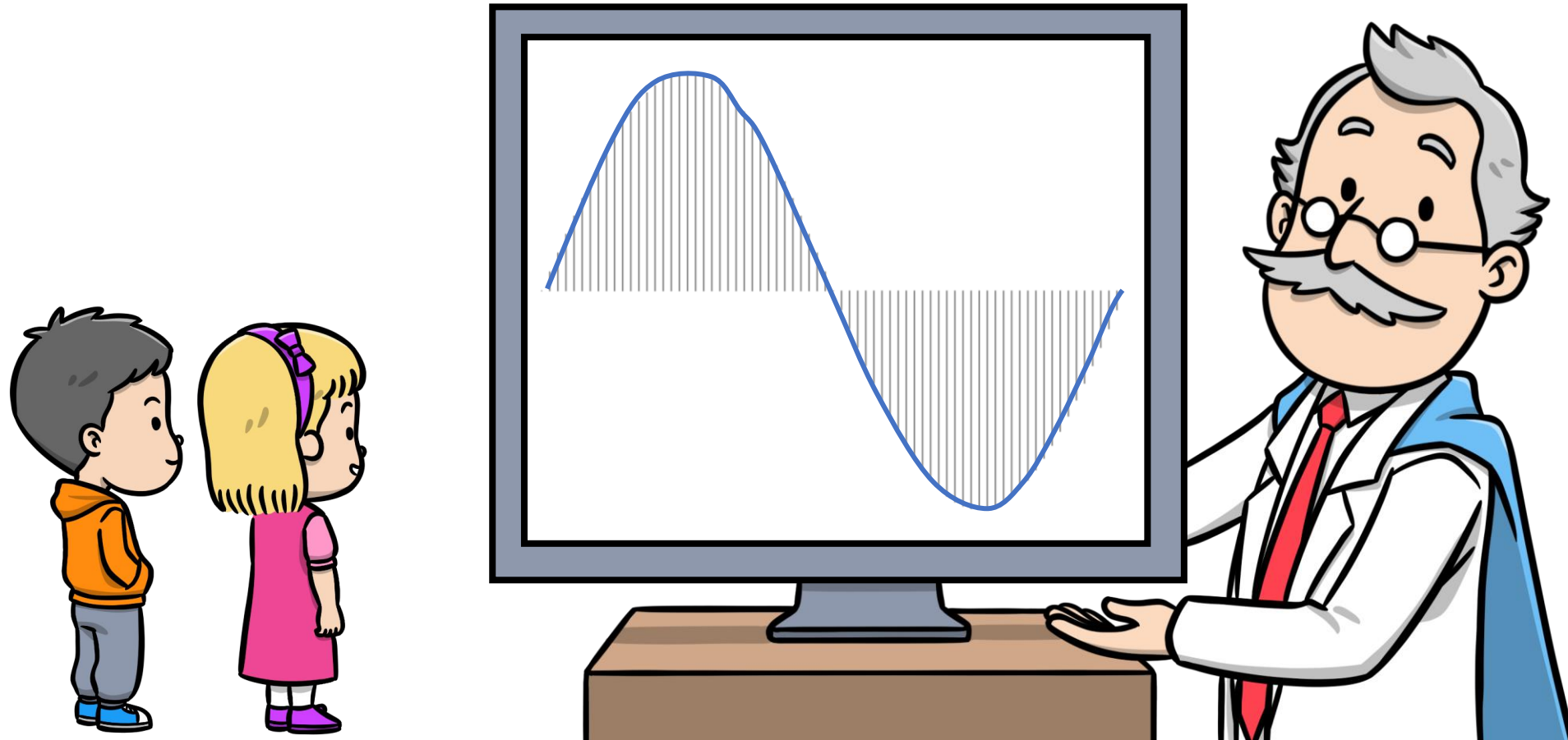
if (isPrime)
{
    println(n, " is prime");
}

else
{
    println(n, " is not prime");
}
```



# Exercise: Graphing sine function

- Let's try another math exercise!
- We want to draw the sine function using vertical lines



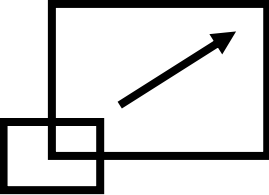
# $\sin(x)$ graph

- Graph of  $\sin(x)$  when  $x$  goes from 0 to 360 degrees
- Function has maximum value 1 for  $x = 90$  and minimum value -1 for  $x = 270$





# Scaling the graph

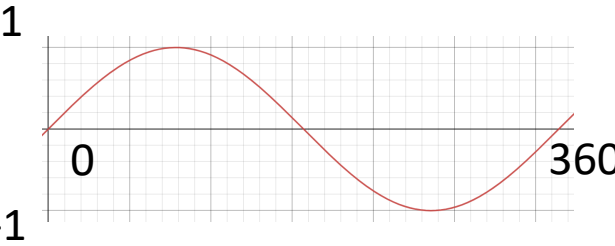
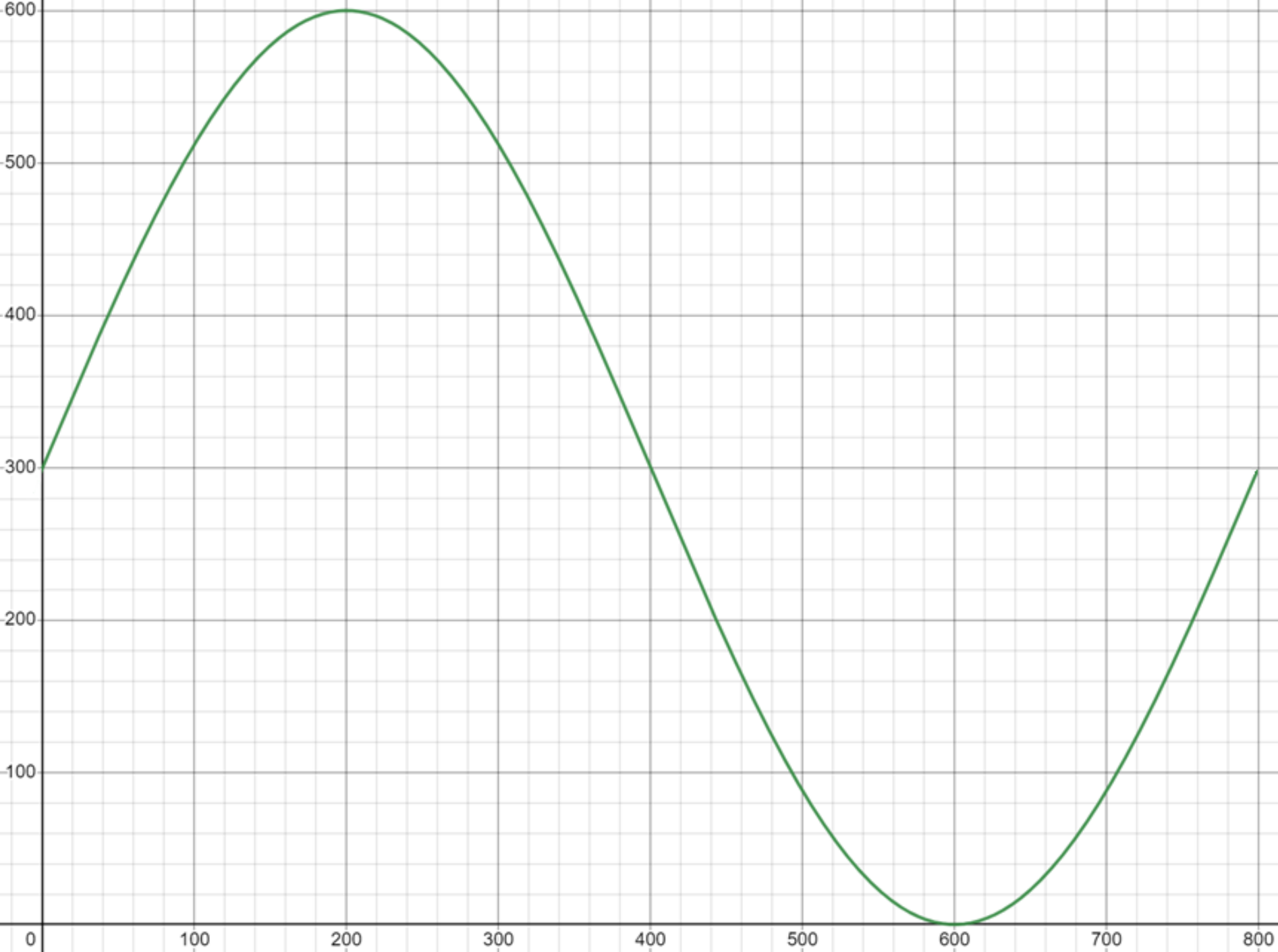


**On horizontal:**

$$0 - 360 \rightarrow 0 - 800$$

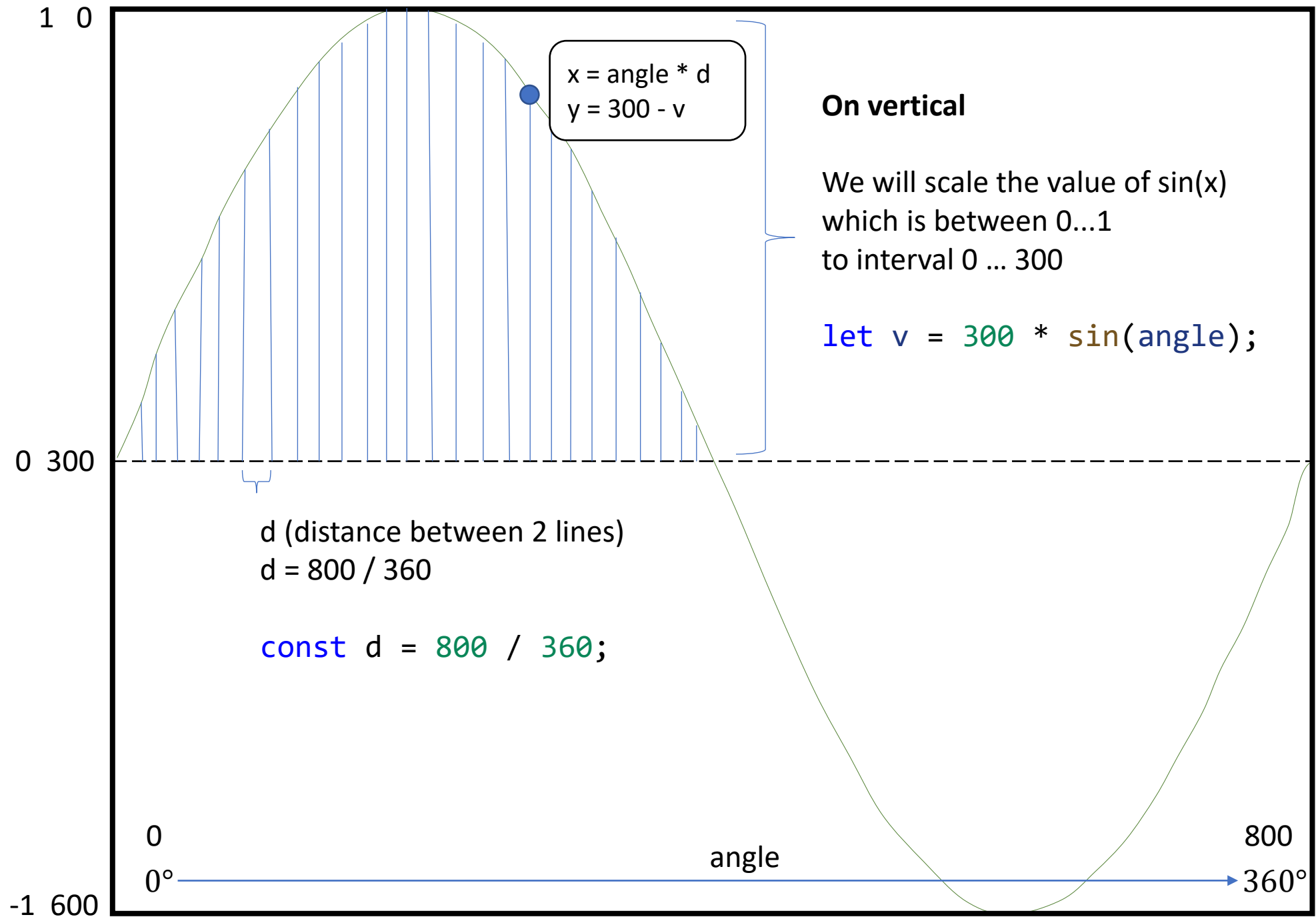
**On vertical:**

$$-1 - 1 \rightarrow 0 - 600$$



## Drawing algorithm

- We will decide how many pixels  $d$  we want between lines
- We will cycle using a for loop through all  $angle$  values between 0 ... 360
- We will translate the angle value to an x value:  $x = angle * d$
- We will calculate  $\sin(angle)$  and then translate this to canvas value  $v$ :  
 $v = 300 * \sin(angle)$
- The y coordinate is:  
 $y = 300 - v$



# Exercise: Graphing sine function (solution)

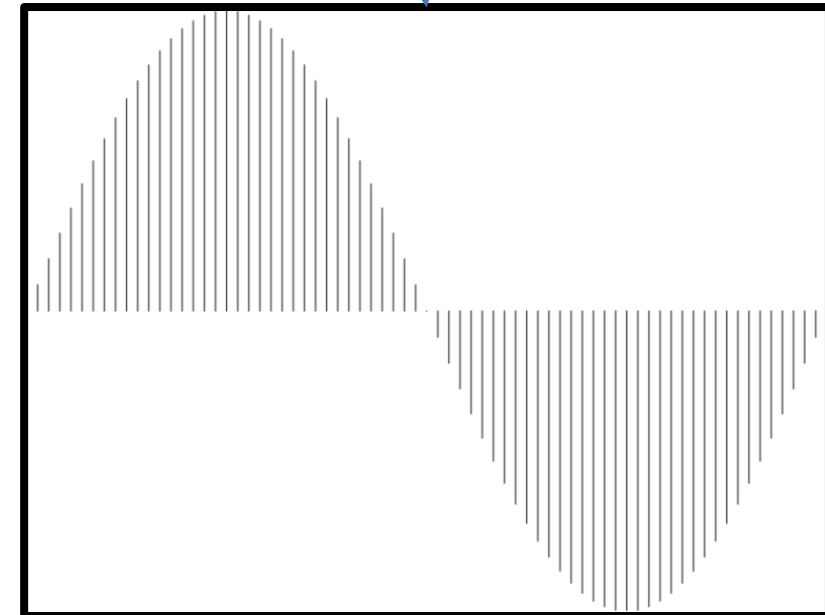
- This is the complete program. On the right you can see the lines taking shape of a sine graph.



```
const d = 800 / 360;
for(let angle = 0; angle < 360; angle += 5)
{
    // sin returns values between -1 and 1
    // we multiply with 300 to stretch on vertical
    let v = 300 * sin(angle);

    let x = angle * d;
    let y = 300;

    line(x, y, x, y - v);
}
```



# Exercise: Graphing sine function (bigger interval)

- Sine is a periodic function. We can update your program to choose how many periods we want to display. Try changing the value of  $n$  and re-run the program (the one on the right).

```
const d = 800 / 360;

for(let angle = 0; angle < 360; angle += 5)
{
  let v = 300 * sin(angle);

  let x = angle * d;
  let y = 300;

  line(x, y, x, y - v);
}
```

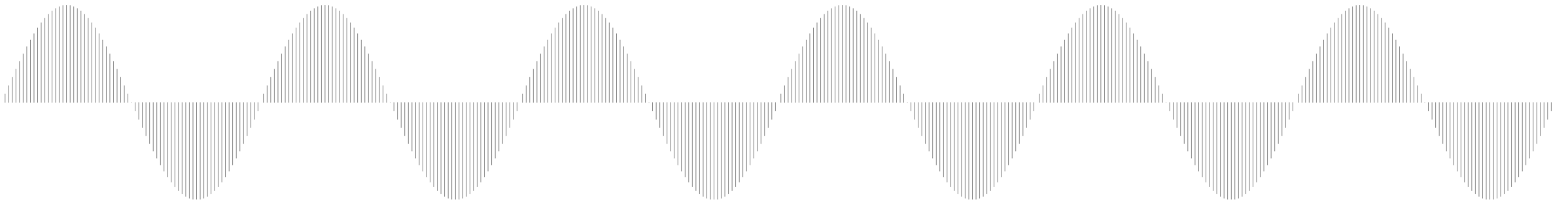


```
const n = 1;
const d = 800 / (360 * n);

for(let angle = 0; angle < 360 * n; angle += 5)
{
  let v = 300 * sin(angle);

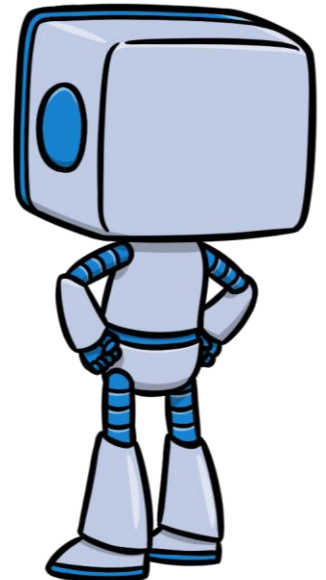
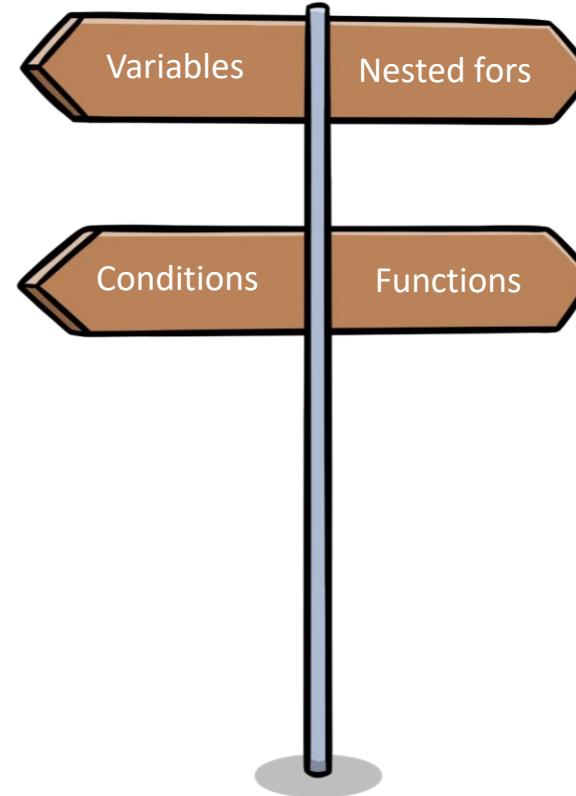
  let x = angle * d;
  let y = 300;

  line(x, y, x, y - v);
}
```



# Next steps

- Try to code in your spare time! Don't be frustrated if you'll encounter errors. Coding required **lots of practice** until you get comfortable writing programs.
- When working on a program, try to **run the program** from time to time to avoid accumulation of errors.
- In the next lesson, we'll revisit the *for* loops (doing mostly **nested *fors***), then we'll learn about **functions**.





## Chapter VIII – Nested for loops

- for loop recap
- Introducing nested for loops
  
- **Exercise:** Multiplication tables 1 to 10
- **Exercise:** Grid of concentric circles
- **Exercise:** Brick pattern
- **Exercise:** Maze pattern
- **Exercise:** Grid of animated sprites

Declare and initialize a variable

Stop condition. For will work as long as this condition is true.

Variable update. Usually increment or decrement.

```
for ( let i = 0; i < 10; i++ )
```

FROM

TO

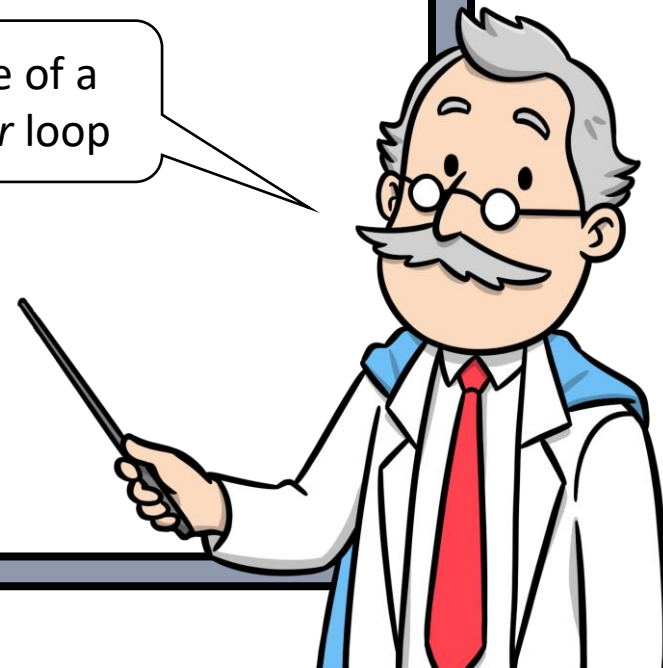
STEP

```
{  
  ...  
}
```



Structure of a typical *for* loop

Code block (block on JavaScript statements) that will be repeated as long as the Stop Condition in *for* is true.



# Nested *for* loops



```
for( let i = 0; i < 10; i++ )  
{  
  for( let j = 0; j < 10; j++ )  
  {  
    println("Hello");  
  }  
}
```

Outer *for* loop  
(uses variable *i*)

Inner *for* loop  
(uses variable *j*)

The inner loop runs many times inside the outer loop in accordance with outer loop configuration.

Question: What do you think is the output of this program?



# Nested *for* loops observations

- Use **different variables** for the outer and inner *for* loops. A typical generic option is to use *i* and *j*. Of course, it is recommended to use the variable name that makes sense for your program.
- In the following examples, the *inner for loop* is executed is executed for each value of the *i* variable in the outer for loop
- The *println* line in the *inner for loop* is therefore executed 100 times (10 \* 10). You can see this better in the second program where we display the values of variables *i* and *j*. Notice that *i* gets incremented only after a complete inner loop is executed.

```
for( let i = 0; i < 10; i++ )  
{  
    for( let j = 0; j < 10; j++ )  
    {  
        println("Hello");  
    }  
}
```



```
for( let i = 0; i < 10; i++ )  
{  
    for( let j = 0; j < 10; j++ )  
    {  
        println("i=", i, " j=", j);  
    }  
}
```



# Exercise: Multiplication tables 1 to 10

- Let's take the code of the multiplication table with 2 that we wrote in the last lesson and add it in the code block of an outer for.
- Adjust the code to make use of the outer for variable



```
const n = 2;  
  
for(let i = 1; i <= 10; i++)  
{  
    let prod = n * i;  
    println(n, " * ", i, " = ", prod);  
}
```

Multiplication table with 2 (code from previous lesson)

```
for(let no = 1; no <= 10; no++)  
{  
    ...  
}
```

Solution on next slide...

# Exercise: Multiplication tables 1 to 10 (solution)



```
for(let no = 1; no <= 10; no++)  
{  
  for(let i = 1; i <= 10; i++)  
  {  
    let prod = no * i;  
    println(no, " * ", i, " = ", prod );  
  }  
  println("");  
}
```



Use variable `no` inside the inner for loop to vary the first number.

- There are 2 statements inside the outer *for*:
- the inner *for*
  - the *println* line

Use an empty print to create a separation line between tables.

```
1 * 1 = 1  
1 * 2 = 2  
1 * 3 = 3  
1 * 4 = 4  
1 * 5 = 5  
1 * 6 = 6  
1 * 7 = 7  
1 * 8 = 8  
1 * 9 = 9  
1 * 10 = 10
```

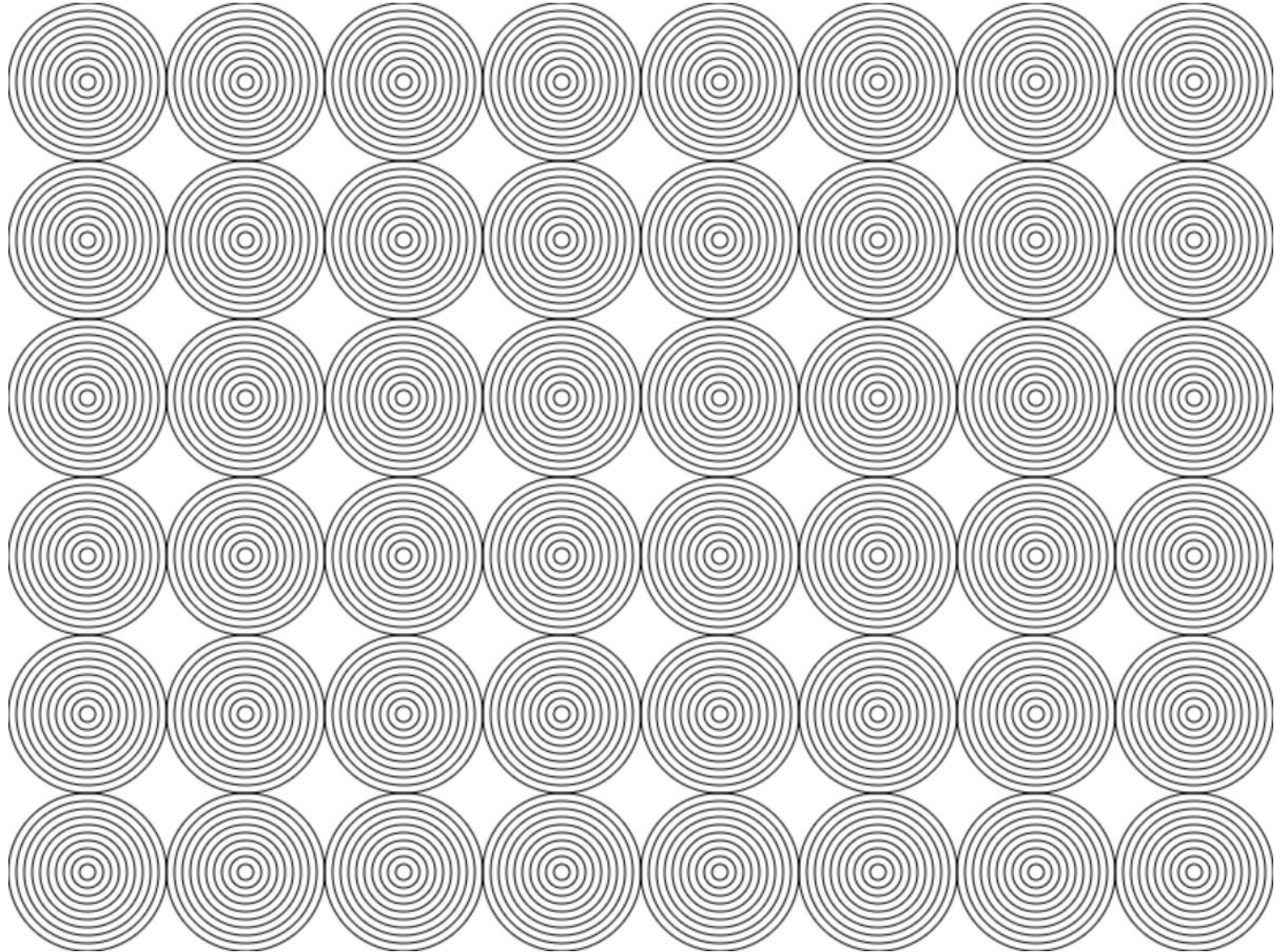
```
2 * 1 = 2  
2 * 2 = 4  
2 * 3 = 6  
2 * 4 = 8  
2 * 5 = 10
```

...

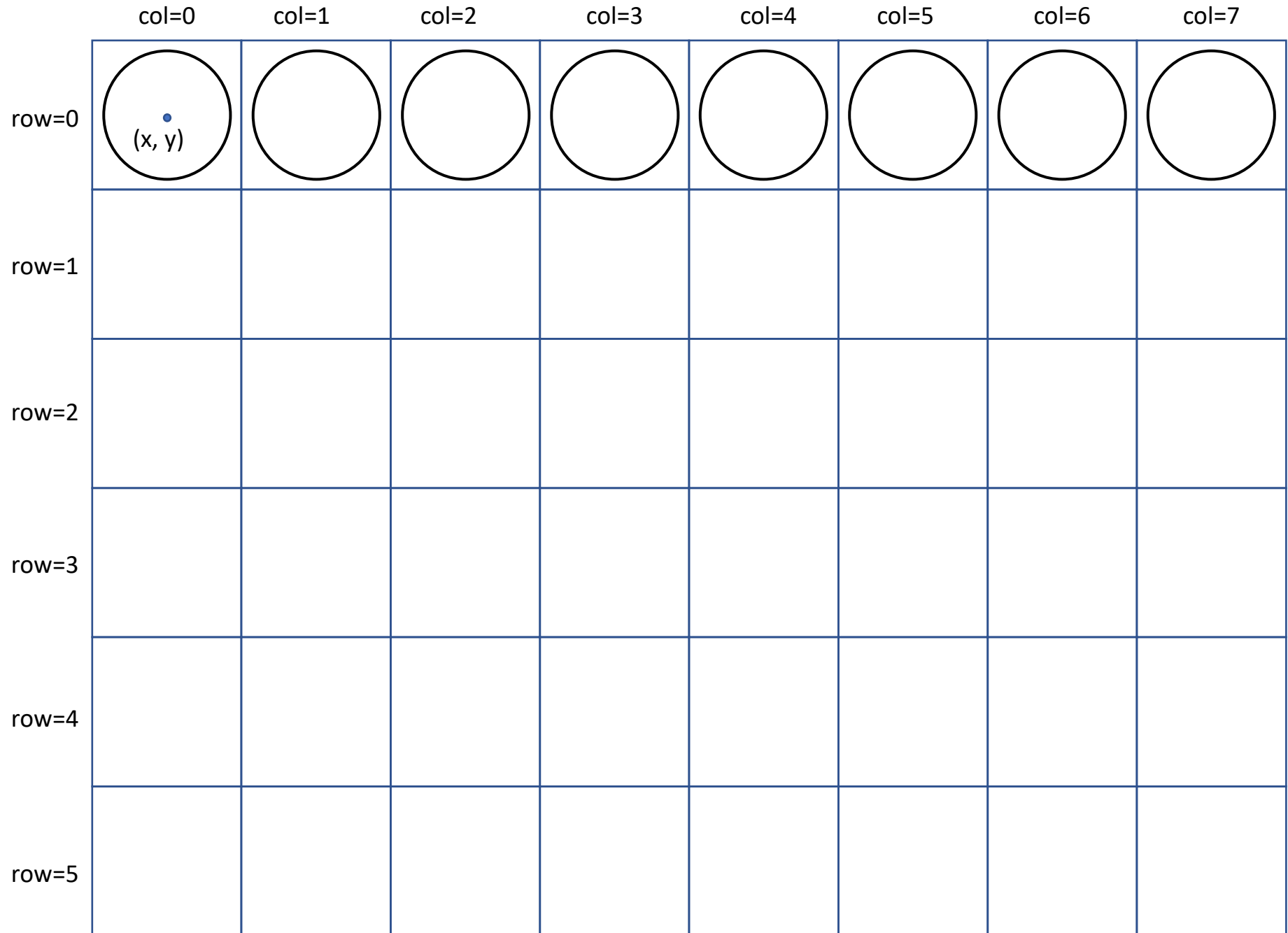
```
10 * 1 = 10  
10 * 2 = 20  
10 * 3 = 30  
10 * 4 = 40  
10 * 5 = 50  
10 * 6 = 60  
10 * 7 = 70  
10 * 8 = 80  
10 * 9 = 90  
10 * 10 = 100
```

# Exercise: Grid of concentric circles

- Let's draw a grid of 8 x 6 discs
- Each disc is a series of concentric circles
- Do you want to attempt the exercise yourselves, or do you want to see the solution?



- We need to place the circles in an imaginary 8 columns x 6 rows grid
- Instead of discs, we will first draw regular circles
- For each circle we need to determine the  $(x, y)$  coordinates of the center based on the row and column position
- We'll draw the circles row by row starting with the top row till the bottom



- Let's display the first row of circles using a single *for loop* (we'll calculate the x and y of circles inside the *for loop*)
- We also take the opportunity to set some variables. Instead of hardcoding the number of rows and columns, we calculate them based on the square size. In this way, we can vary the number of circles via a single update.

Type and  
run this  
program

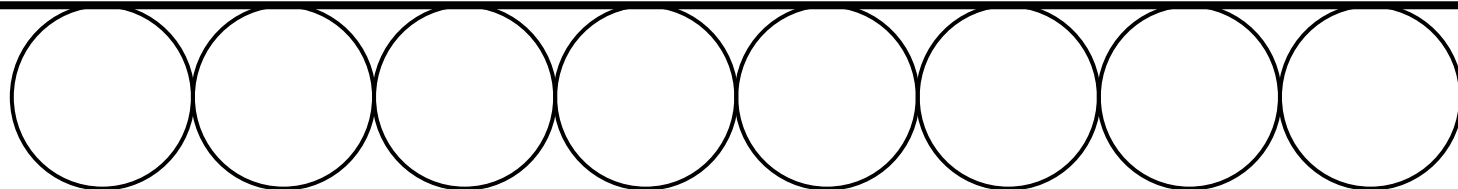
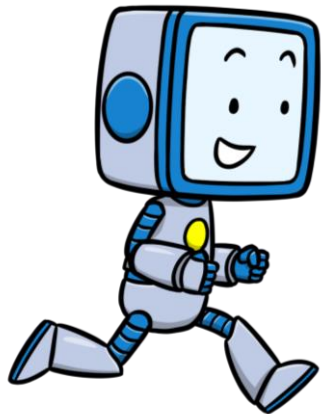


```
// Square size
const squareSize = 100;

// Calculates the number of rows and columns that fit on the screen
const rows = 600 / squareSize;
const cols = 800 / squareSize;

// Loop through all the columns
for(let col = 0; col < cols; col++)
{
    let x = squareSize / 2 + col * squareSize; // 50 150 250 350 ...
    let y = squareSize / 2;

    circle(x, y, squareSize / 2);
}
```



- We'll *wrap* now the for loop that draws a line of circles in an *outer for* that will *loop* through all the rows on the canvas
- We'll also use the variable of the outer for (*row*) inside the *inner for* to calculate the *y* coordinate of each row of circles

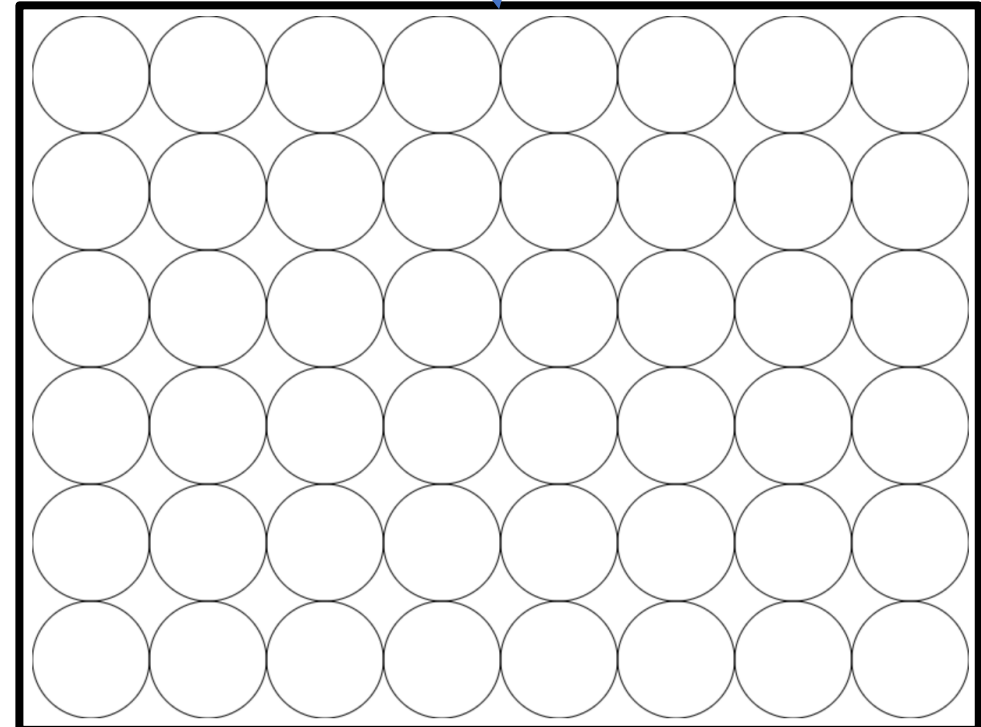
```
const squareSize = 100;
const rows = 600 / squareSize;
const cols = 800 / squareSize;

// Loop through all the rows
for(let row = 0; row < rows; row++)
{
  // For each row, loop through all the columns
  for(let col = 0; col < cols; col++)
  {
    let x = squareSize / 2 + col * squareSize;
    let y = squareSize / 2 + row * squareSize;

    circle(x, y, squareSize / 2);
  }
}
```



Nested for loops are great  
for 2D operations



# Exercise: Grid of concentric circles (full solution)

```
// Square size. Change this number and re-run the program
const squareSize = 100;

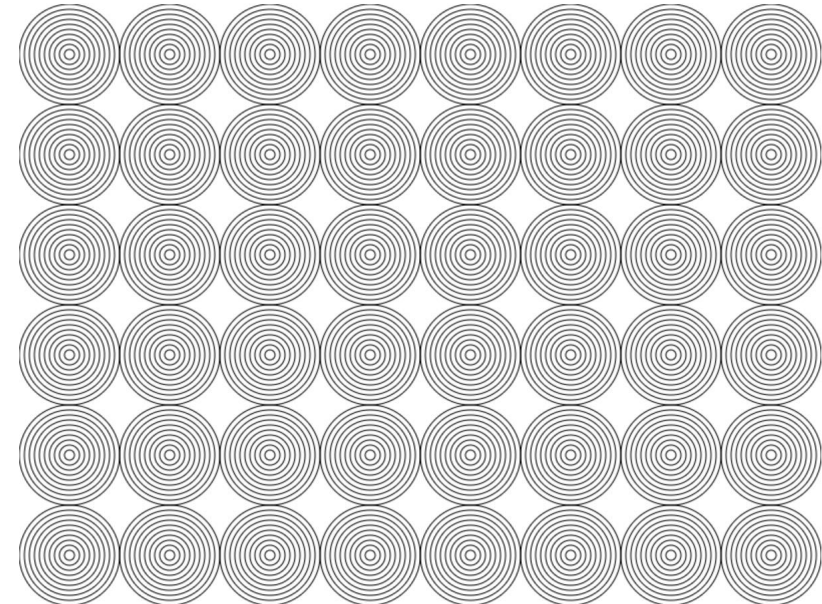
// Calculates the number of rows and columns that fit on the screen
const rows = 600 / squareSize;
const cols = 800 / squareSize;

// Loop through all the rows
for(let row = 0; row < rows; row++)
{
    // For each row, loop through all the columns
    for(let col = 0; col < cols; col++)
    {
        let x = squareSize / 2 + col * squareSize;
        let y = squareSize / 2 + row * squareSize;

        // For each circle, draw inner circles (concentrical)
        for(let r = squareSize / 2; r > 0; r -= 5)
        {
            circle(x, y, r);
        }
    }
}
```



- To draw discs (e.g. concentric circles) instead of regular circles, we place yet another for loop inside the inner one.
- Our code contains now three *nested for loops*!
- Note: The performance of your program may degrade if you use too many nested for loops.





# Exercise: Grid of circles (with a single *for loop*)

- Did you know that you can also use a single *for loop* to draw the grid of circles?
- The version on the right, uses *if* conditions to reset *x* and increase the *y* coordinate when a row is completed (we dropped concentric circles from program)

```
const rows = 6;
const cols = 8;

// Loop through all the rows
for(let row = 0; row < rows; row++)
{
    // For each row, loop through the columns
    for(let col = 0; col < cols; col++)
    {
        let x = 50 + col * 100;
        let y = 50 + row * 100;

        circle(x, y, 50);
    }
}
```

Nested for version (this is our previous program simplified)

Type in this single *for* version. It is good for recapping *ifs*.



```
// Number of circles
let n = 6 * 8;

let x = 50;
let y = 50;

for(let i = 0; i < n; i++)
{
    circle(x, y, 50);

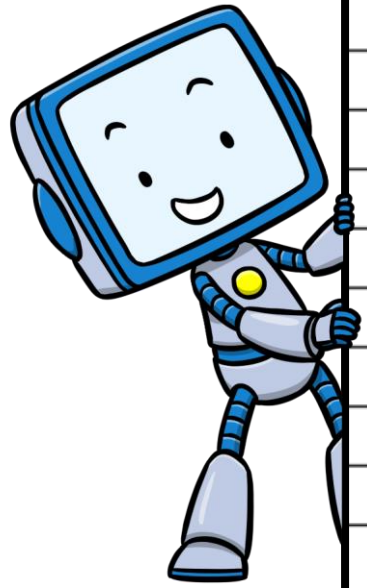
    x += 100;

    if (x > 800)
    {
        x = 50;
        y += 100;
    }

    if (y > 600)
    {
        break;
    }
}
```

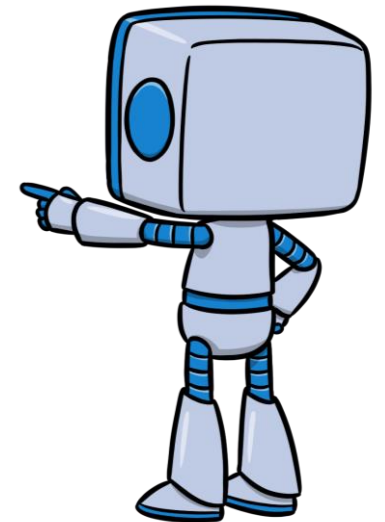
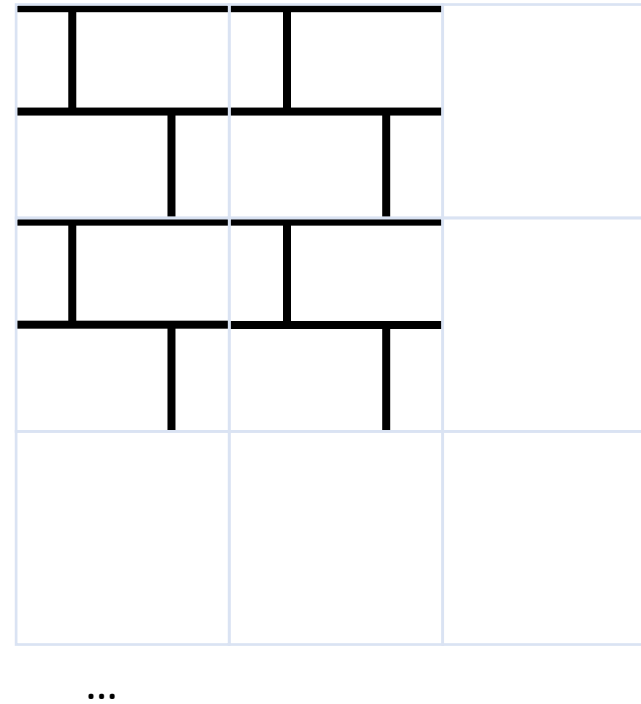
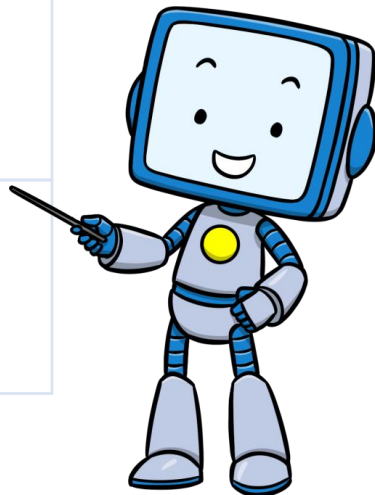
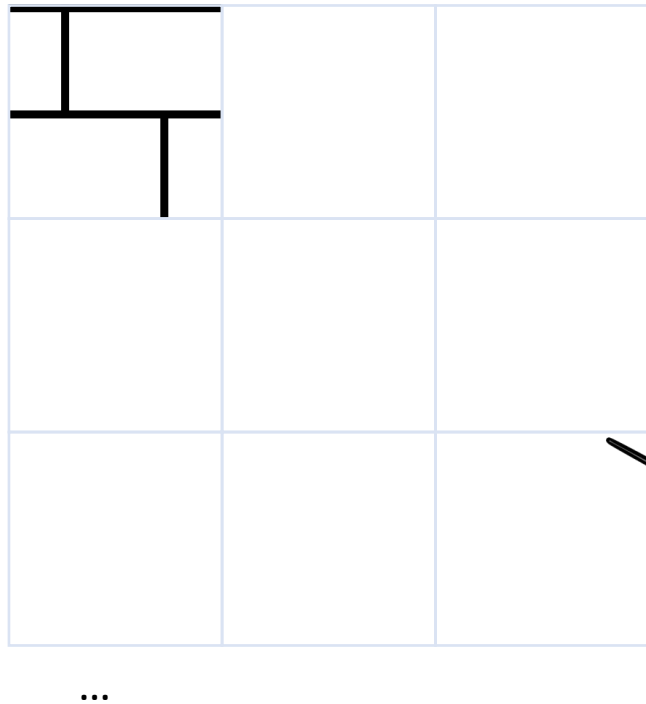
# Exercise: Brick pattern

- Let's draw now a brick pattern
- We can achieve this easily with a *nested for loop*



# Exercise: Brick pattern (planning)

- We will also *divide* the canvas in rows and columns
- Instead of drawing a circle inside each imaginary square, we will draw a line pattern of 4 lines
- When these patterns will repeat in the adjacent cells, a brick pattern will emerge.



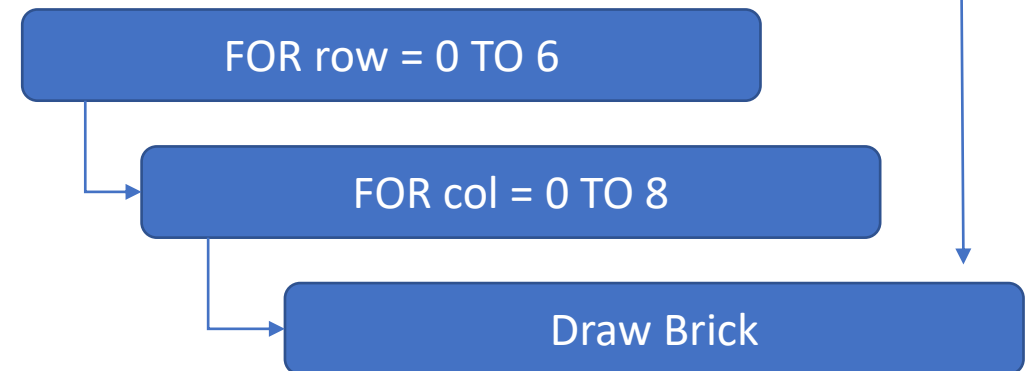
- Let's write first the code that draws a single brick

```
const squareSize = 50;

let x = 100;
let y = 100;

// Draw a single brick...
line(x, y, x + squareSize, y);
line(x, y + squareSize / 2, x + squareSize, y + squareSize / 2);
line(x + squareSize / 4, y, x + squareSize / 4, y + squareSize / 2);
line(x + 3 * squareSize / 4, y + squareSize / 2, x + 3 * squareSize / 4, y + squareSize);
```

- After we test and make sure that the code for drawing a brick is running, we will *wrap* the code in a nested for (like in previous exercise)
- The outer for will loop on rows and the inner for on columns



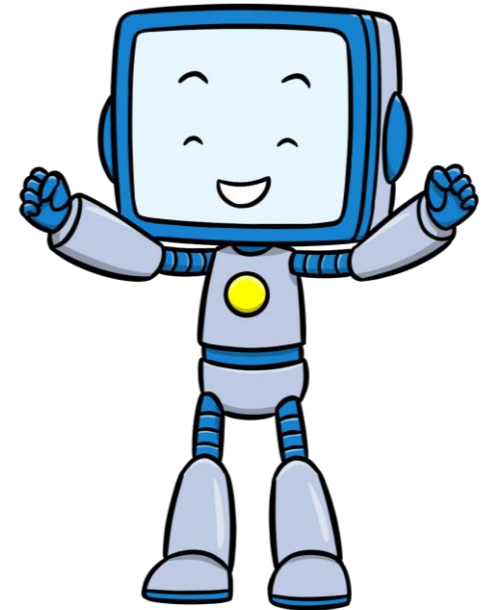
# Exercise: Brick pattern (full solution)

```
// Square size. Change this number and re-run the program
const squareSize = 100;

// Calculates the number of rows and columns that fit on the screen
const rows = 600 / squareSize;
const cols = 800 / squareSize;

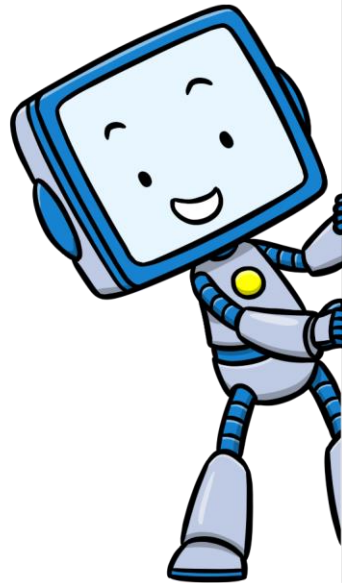
// Loop through all the rows
for(let row = 0; row < rows; row++)
{
  // For each row, loop through all the columns
  for(let col = 0; col < cols; col++)
  {
    let x = col * squareSize;
    let y = row * squareSize;

    // Draw brick...
    line(x, y, x + squareSize, y);
    line(x, y + squareSize / 2, x + squareSize, y + squareSize / 2);
    line(x + squareSize / 4, y, x + squareSize / 4, y + squareSize / 2);
    line(x + 3 * squareSize / 4, y + squareSize / 2, x + 3 * squareSize / 4, y + squareSize);
  }
}
```



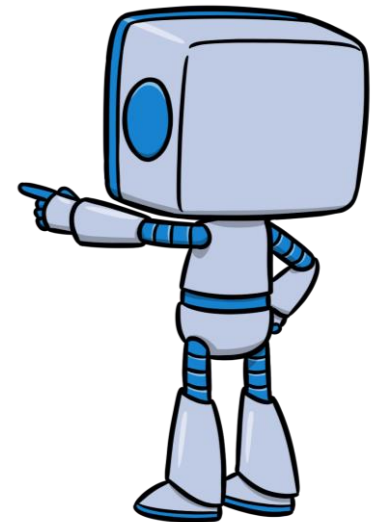
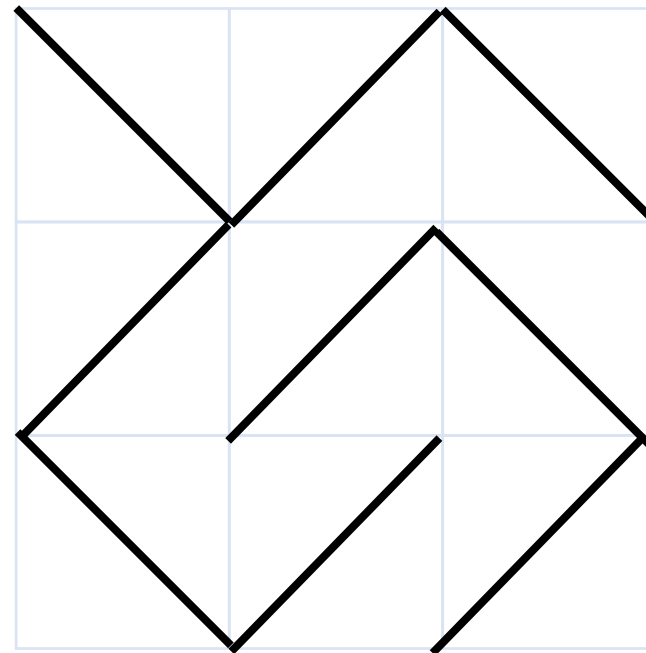
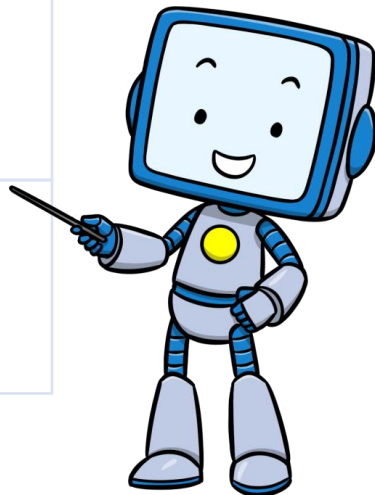
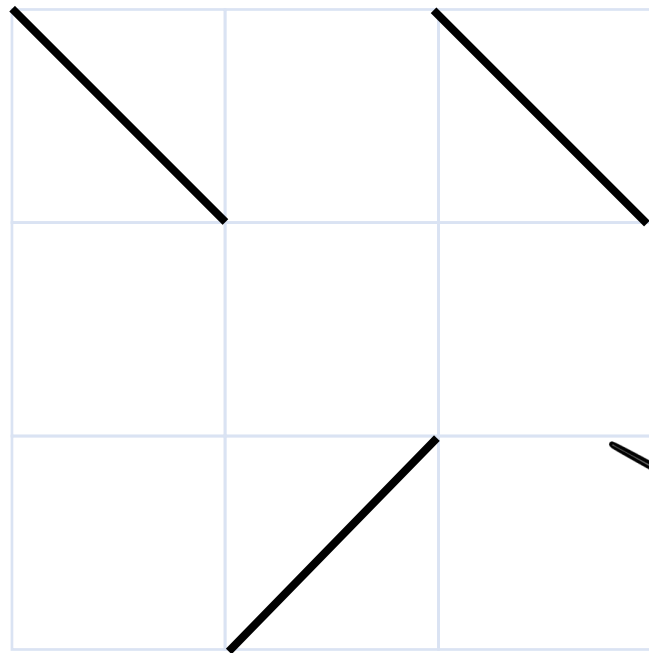
# Exercise: Maze pattern

- This exercise takes the brick pattern one step forward
- The pattern that you was popular in the early days of microcomputers, especially on the Commodore 64 computer
- Let's see what it takes to implement it in JavaScript!



# Exercise: Maze pattern (planning)

- We will also *divide* the canvas in rows and columns (e.g. cells)
- In each cell we will draw a random diagonal line (either left to right or right to left).
- The lines will appear as they are connected to the ones from the adjacent cells, therefore a maze pattern will emerge



# Nested for loop to process the imaginary rows and columns grid

```
// Square size. Change this number and re-run
const squareSize = 25;


// Calculates the number of rows and columns
const rows = 600 / squareSize;
const cols = 800 / squareSize;

// Loop through all the rows
for(let row = 0; row < rows; row++)
{
    // For each row, loop through all the columns
    for(let col = 0; col < cols; col++)
    {
        let x = col * squareSize;
        let y = row * squareSize;

        rect(x, y, squareSize, squareSize);
    }
}
```



- The program uses a nested for to loop over rows and columns
- In the inner for we calculate x and y of each imaginary square
- Let's temporarily draw a rectangle (e.g. square) at those coordinates to see if the grid is display as intended

When done typing, run the program 



# The pattern drawing code

- We will now replace the “rect” instruction with a few lines of code that will randomly draw diagonal lines (either \ or /) based on a random number.
- In the code below n is a decimal random number between 0 and 1. There is 50% probability it will be less than 0.5 and 50% probability that is above 0.5 (therefore the if condition). `random()` is a built-in function that gives random numbers.

```
// n random between 0 and 1...
let n = random();
if (n < 0.5)
{
  // line \
  line(x, y, x + squareSize, y + squareSize);
}
else
{
  // line /
  line(x + squareSize, y, x, y + squareSize);
}
```

```
...
for(let row = 0; row < rows; row++)
{
  for(let col = 0; col < cols; col++)
  {
    let x = col * squareSize;
    let y = row * squareSize;

    rect(x, y, squareSize, squareSize);
  }
}
```



```
// Square size. Change this number and re-run
const squareSize = 25;

// Calculates the number of rows and columns
const rows = 600 / squareSize;
const cols = 800 / squareSize;

// Loop through all the rows
for(let row = 0; row < rows; row++)
{
    // For each row, loop through all the columns
    for(let col = 0; col < cols; col++)
    {
        let x = col * squareSize;
        let y = row * squareSize;

        // n random between 0 and 1...
        let n = random();

        if (n < 0.5)
        {
            // line \
            line(x, y, x + squareSize, y + squareSize);
        }
        else
        {
            // line /
            line(x + squareSize, y, x, y + squareSize);
        }
    }
}
```

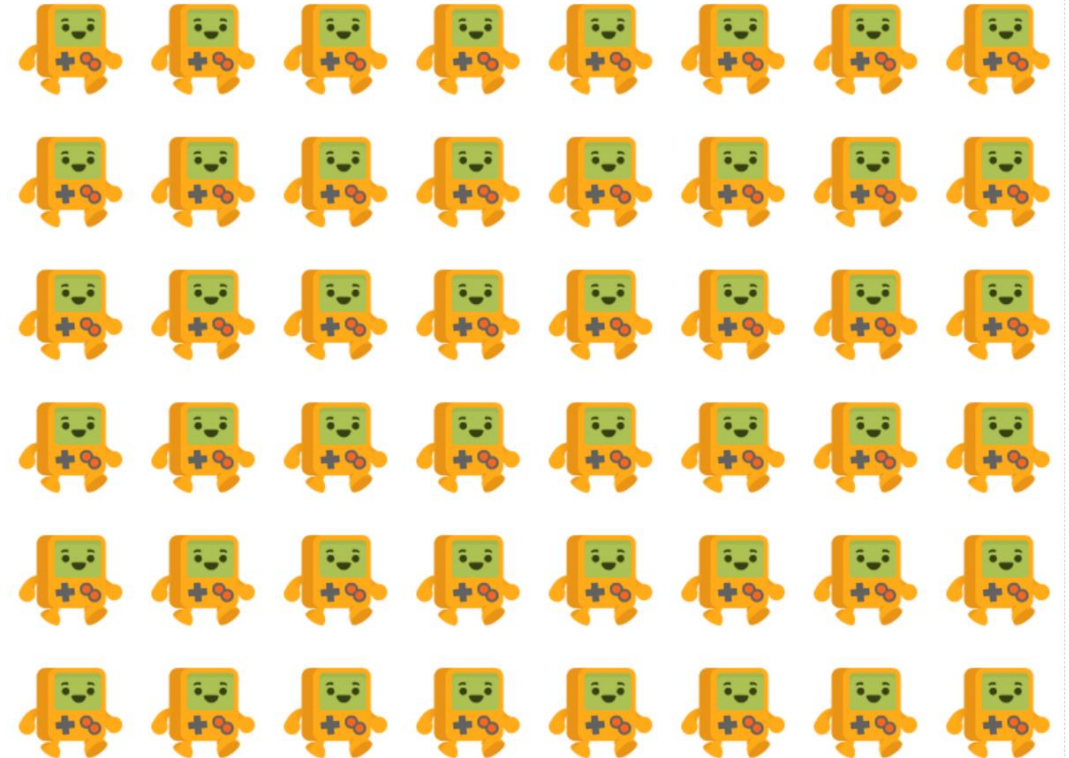
This is the complete listing of the maze program!



# Exercise: Lots of sprites!

- Remember the first hour of code when we dragged and dropped a sprite in the code area? That action used to create `sprite` instructions for us.
- Let's try to put that instruction in a nested for and try to create a grid of sprites! Type in the program to see the effect.

```
let rows = 6;  
let cols = 8;  
  
for(let row = 0; row < rows; row++)  
{  
  for(let col = 0; col < cols; col++)  
  {  
    let x = 50 + col * 100;  
    let y = 50 + row * 100;  
  
    sprite('game.walk', x, y, 0.5);  
  }  
}
```





<https://codeguppy.com>

Free coding platform