P.PORTO

POLITÉCNICO
DO PORTO
ESMAD

COMPUTAÇÃO GRÁFICA
TSIW

Syllabus

- Styling:
  - Colors
  - Transparency
- Paths
- Shapes in paths:
  - Lines
  - Arcs
  - Curves
- Text
- Animation cycle

Styling

COLORS

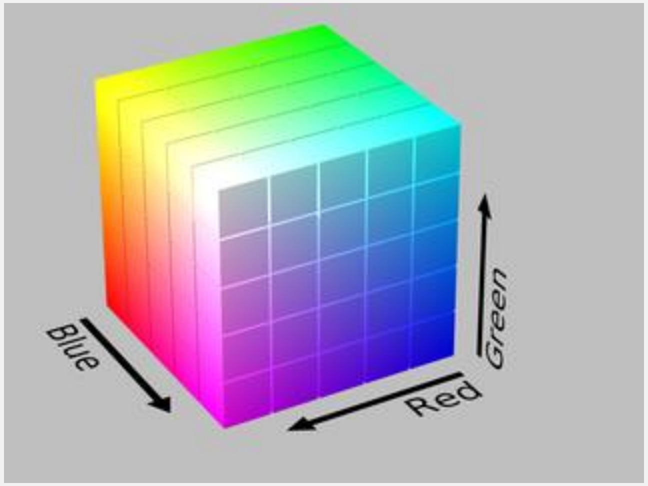
- Use properties **`fillStyle`** and **`strokeStyle`** to define an object's color
- Possible values:
  - Color name (EN) from CSS (https://developer.mozilla.org/en-US/docs/Web/CSS/named-color)
  - Hexadecimal values
  - Decimal values, using function `rgb()`

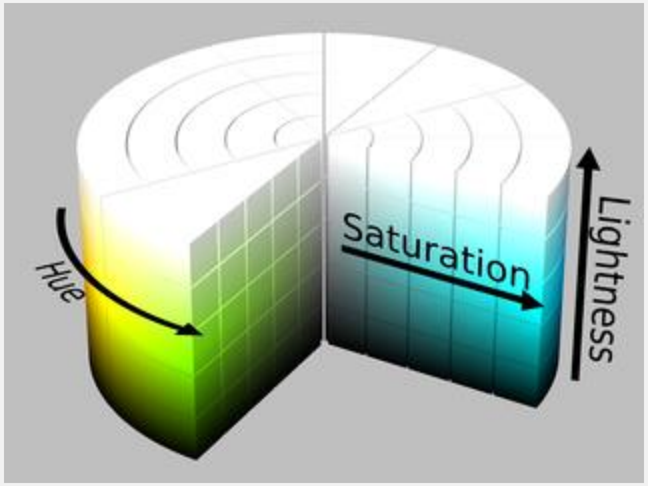
Color	Name	HEX	HEX (abreviated)	RGB
🟥	red	#FF0000	#F00	rgb(255,0,0)
🟩	green	#00FF00	#0F0	rgb(0,255,0)
🟦	blue	#0000FF	#00F	rgb(0,0,255)

Styling

COLORS

- Only the objects drawn **after a color change** are affected
- Current browsers support CSS3 colors and allow the use of color spaces other than RGB: RGBA, HSL, and HSLA





Styling

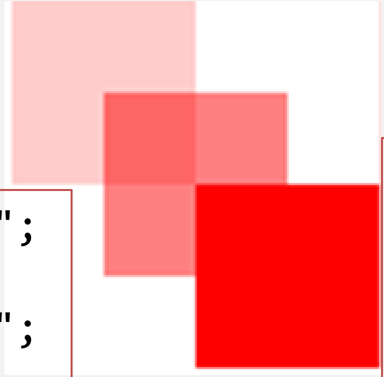
TRANSPARENCY/OPACITY

- A (*alpha*): float value to define opacity

  0: totally transparent

  1: totally opaque

- Opacity can also be defined using property **`globalAlpha`**



```
ctx.fillStyle = "rgba(255,0,0,0.2)";
ctx.fillRect(0,0,50,50);
ctx.fillStyle = "rgba(255,0,0,0.5)";
ctx.fillRect(25,25,50,50);
ctx.fillStyle = "rgba(255,0,0,1)";
ctx.fillRect(50,50,50,50);
```

```
ctx.fillStyle = "red";
ctx.globalAlpha = 0.2;
ctx.fillRect(100,0,50,50);
ctx.globalAlpha = 0.5;
ctx.fillRect(125,25,50,50);
ctx.globalAlpha = 1;
ctx.fillRect(150,50,50,50);
```

Styling

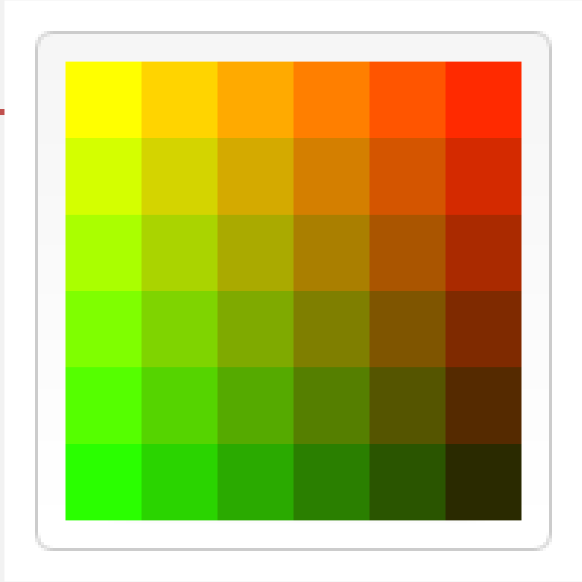
COLORS & TRANSPARENCY/OPACITY

```
"#f00"             // Hexadecimal RGB value: red
"#00ff00"          // RRGGBB value: green

"rgb(60, 60, 255)"     // RGB as integers: blue
"rgb(100%, 25%, 100%)" // RGB as percentages: purple
"rgba(100%,25%,100%,0.5)"   // Plus alpha 0-1: translucent
"rgba(0,0,0,0)"        // Transparent black

"transparent"          // Synonym for the above

"hsl(60, 100%, 50%)"   // Fully saturated yellow
"hsl(60, 75%, 50%)"    // Less saturated yellow
"hsl(60, 100%, 75%)"   // Fully saturated, lighter
"hsl(60, 100%, 25%)"   // Fully saturated, darker
"hsla(60,100%, 50%, 0.5)"   // 50% opaque
```

Styling

COLORS & TRANSPARENCY/OPACITY

```
const ctx = document.getElementById("canvas").getContext("2d");
  for (let i = 0; i < 6; i++) {
    for (let j = 0; j < 6; j++) {
      ctx.fillStyle = `rgb(${Math.floor(255-42.5*i)}, ${Math.floor(255-42.5*j,)}, 0)`;
      ctx.fillRect(j * 25, i * 25, 25, 25);
    }
  }
```



Paths

- Unlike SVG, Canvas only supports two primitive shapes: **rectangles** and **paths**
  - All other shapes must be created by combining one or more paths
  - However, Canvas API provides an assortment of path drawing functions which make it possible to compose very complex shapes
- Defining a path on Canvas is like **drawing with a pencil**
- **Path** is a sequence of points, lines or curves (subpaths) to be drawn between the start and end points

Paths

STEPS to make shapes using paths:

1. (Re)Start a new path: **`beginPath()`**
2. “Move the pencil”, i.e. create a set of subpaths (lines, arcs, curves)
3. “Paint”, meaning draw the final shape:
   `stroke()` - just the outline
   `fill()`- full form fill
4. (optional) Close the path by drawing a line between the endpoint and start point: **`closePath()`**

Path methods

`beginPath()`

Starts a new path by emptying the list of sub-paths;
Call this method when you want to create a new path.

`closePath()`

Moves back the pen from the back to the beginning of the current sub-path (by drawing a line);
If the shape has already been closed or has only one point, this function does nothing.

Path methods

`moveTo(x,y)`
Moves the starting point of a new sub-path to the **`(x,y)`** coordinates

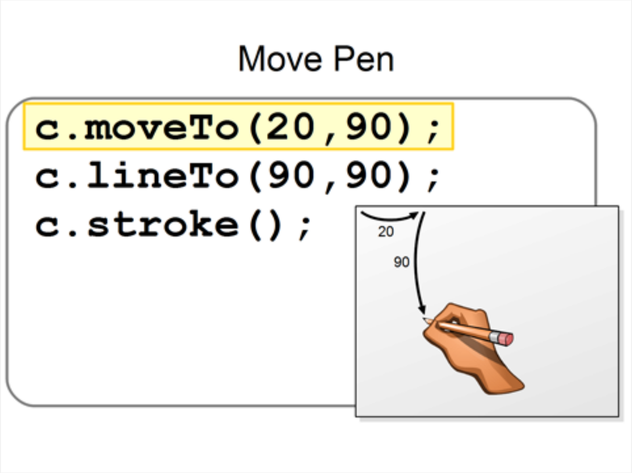
`lineTo(x,y)`
Connects the last point in the current sub-path to the specified **`(x,y)`** coordinates with a straight line

Paths & Lines

- **Lines**:

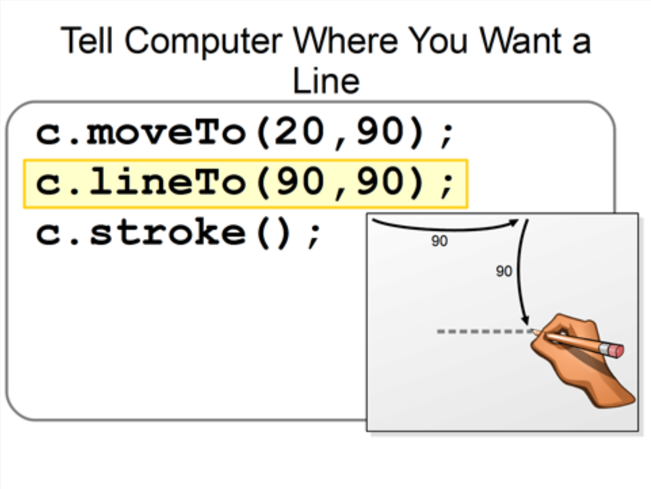
1. moveTo(x,y)

mandatory!!!



Move Pen

```
c.moveTo(20,90);
c.lineTo(90,90);
c.stroke();
```

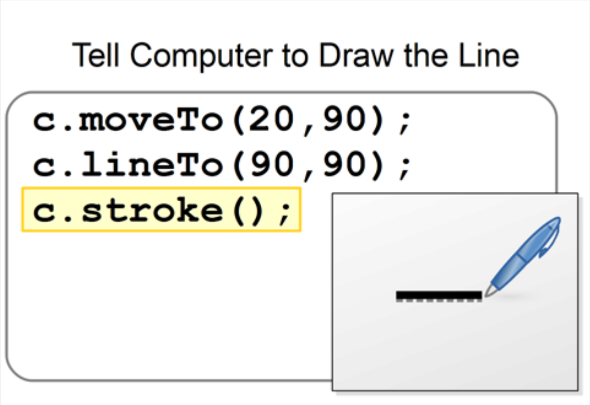


Tell Computer Where You Want a Line

```
c.moveTo(20,90);
c.lineTo(90,90);
c.stroke();
```

2. lineTo(x,y)

3. stroke()

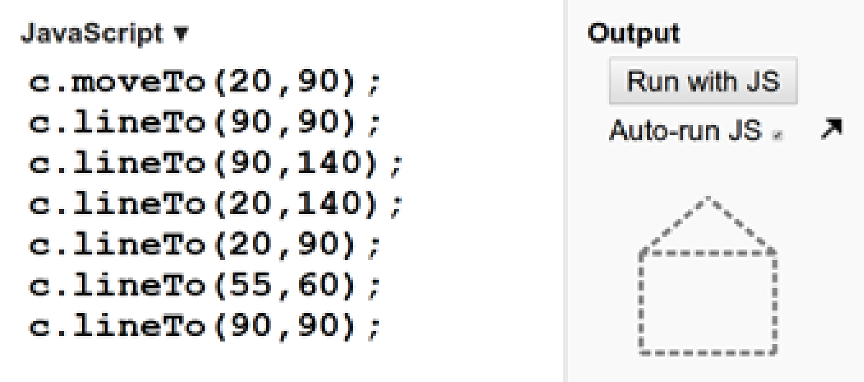


Tell Computer to Draw the Line

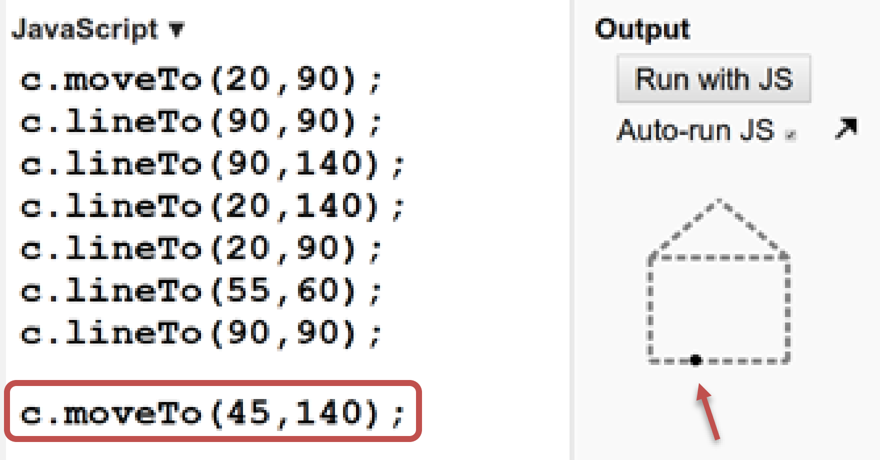
```
c.moveTo(20,90);
c.lineTo(90,90);
c.stroke();
```

Paths & Lines

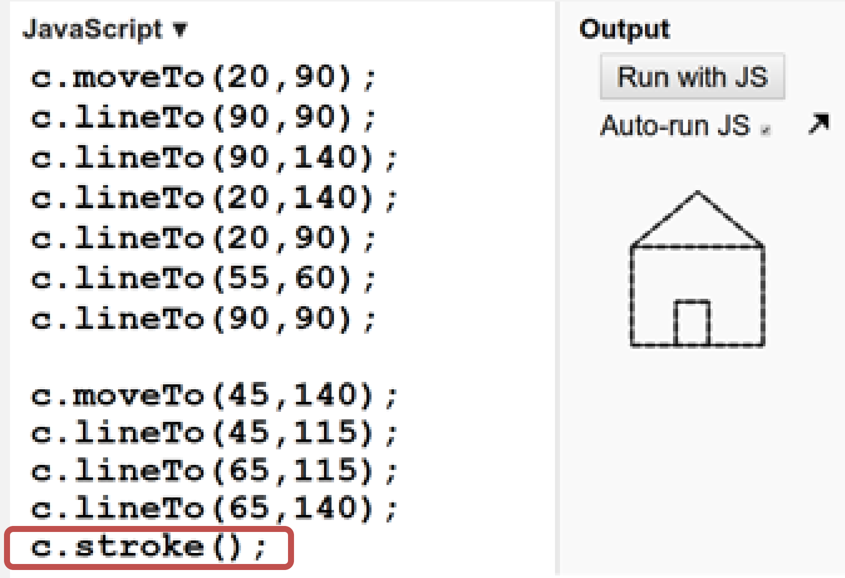
- **Lines**:



```
c.moveTo(20,90);
c.lineTo(90,90);
c.lineTo(90,140);
c.lineTo(20,140);
c.lineTo(20,90);
c.lineTo(55,60);
c.lineTo(90,90);
```



```
c.moveTo(20,90);
c.lineTo(90,90);
c.lineTo(90,140);
c.lineTo(20,140);
c.lineTo(20,90);
c.lineTo(55,60);
c.lineTo(90,90);

c.moveTo(45,140);
```



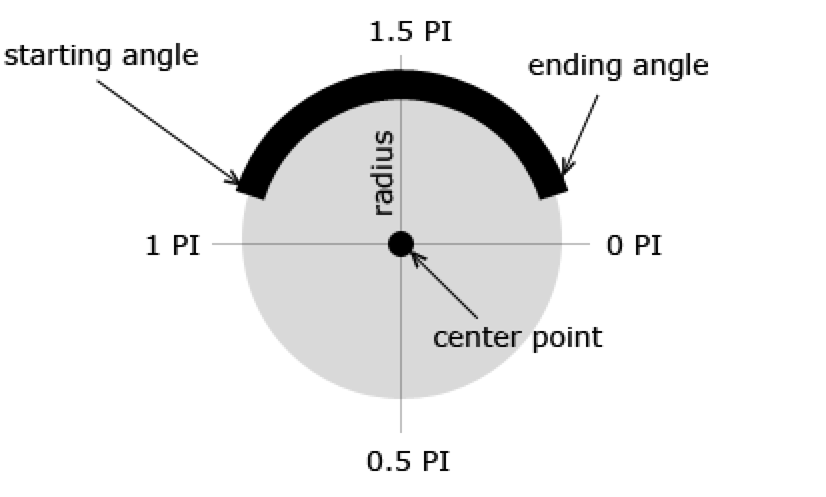
```
c.moveTo(20,90);
c.lineTo(90,90);
c.lineTo(90,140);
c.lineTo(20,140);
c.lineTo(20,90);
c.lineTo(55,60);
c.lineTo(90,90);

c.moveTo(45,140);
c.lineTo(45,115);
c.lineTo(65,115);
c.lineTo(65,140);
c.stroke();
```

Path methods

`arc(cX,cY,r,θi,θf[,dir])`

Adds a circular arc to the current path

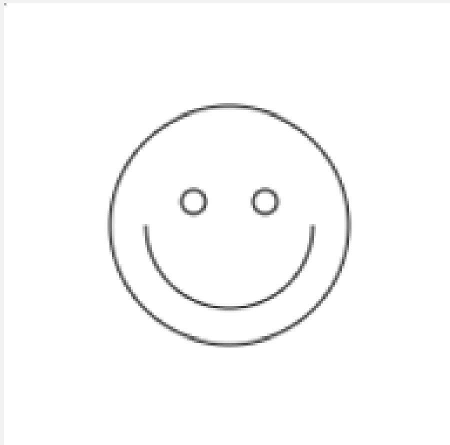
- `cX`, `cY`: center
- `r`: radius
- `θi`: initial angle (radians)
- `θf`: final angle (radians)
- `dir`: direction (optional)
  – default value: false (clockwise)



- Use JS module Math to convert degrees into radians:
  θ_{rad} = `Math.PI / 180 *` θ_{deg}

Paths & Arcs

```
ctx.beginPath();
ctx.arc(75, 75, 50, 0, Math.PI * 2, true); // Outer circle
ctx.moveTo(110, 75);
ctx.arc(75, 75, 35, 0, Math.PI, false);    // Mouth (clockwise)
ctx.moveTo(65, 65);
ctx.arc(60, 65, 5, 0, Math.PI * 2, true);  // Left eye
ctx.moveTo(95, 65);
ctx.arc(90, 65, 5, 0, Math.PI * 2, true);  // Right eye
ctx.stroke();                              // Paint the path border
```



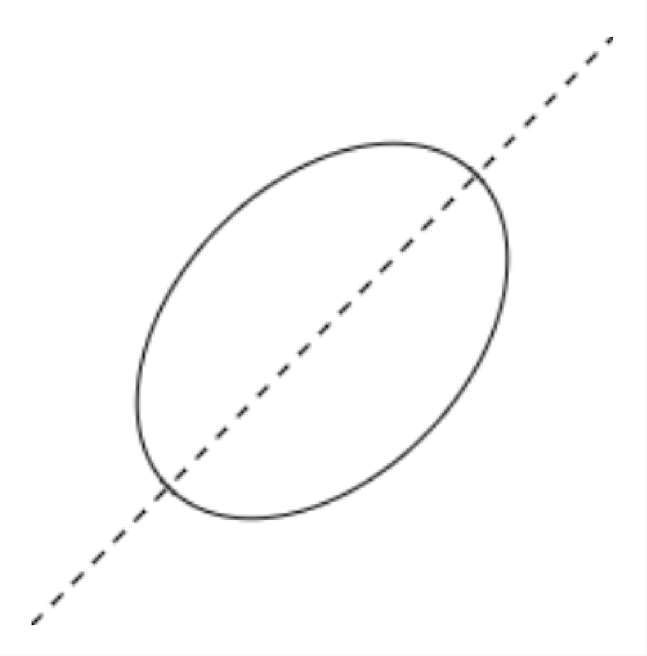
Path methods

```
ellipse(cX,cY, rX,rY, rot, θi,θf [,dir])
```

- `cX`, `cY`: center of the ellipse
- `rX`, `rY`: radius
- `rot`: rotation of the ellipse (radians)
- `θi`: initial angle (radians)
- `θf`: final angle (radians)
- `dir`: direction (optional)
  – default value: false (clockwise)

Paths & Ellipses

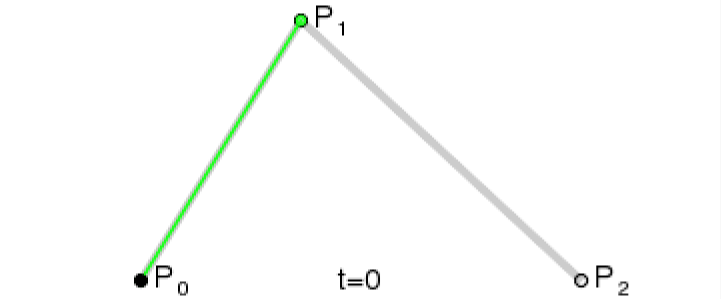
```
// Draw the ellipse
ctx.beginPath();
ctx.ellipse(100, 100, 50, 75, Math.PI / 4, 0, 2 * Math.PI);
ctx.stroke();

// Draw the ellipse's line of reflection
ctx.beginPath();
ctx.setLineDash([5, 5]);
ctx.moveTo(0, 200);
ctx.lineTo(200, 0);
ctx.stroke();
```



Path methods

`quadraticCurveTo(cpX,cpY, x,y)`

- ○ Initial point – P0 – is the current "pencil" position
- ○ **cpX**, **cpY**: control point - P1 - coordinates
- ○ **x**, **y**: final point - P2 - coordinates

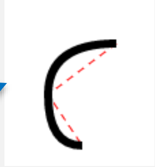


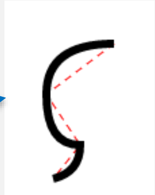
- Interact with the example here

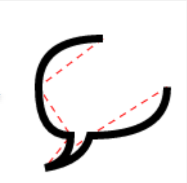
Paths & Curves

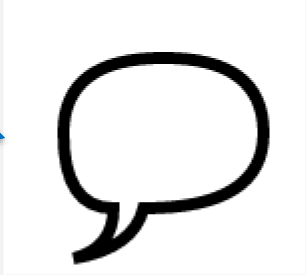
- Example using quadratic curves:

```
ctx.lineWidth = 6;   //increases line width

ctx.beginPath();     //start a new path
ctx.moveTo(75,25);   //sets inicial “pencil” position
ctx.quadraticCurveTo(25,25,25,62.5);
ctx.quadraticCurveTo(25,100,50,100);
ctx.quadraticCurveTo(50,120,30,125);
ctx.quadraticCurveTo(60,120,65,100);
ctx.quadraticCurveTo(125,100,125,62.5);
ctx.quadraticCurveTo(125,25,75,25);

ctx.stroke();        //paint the path border
```





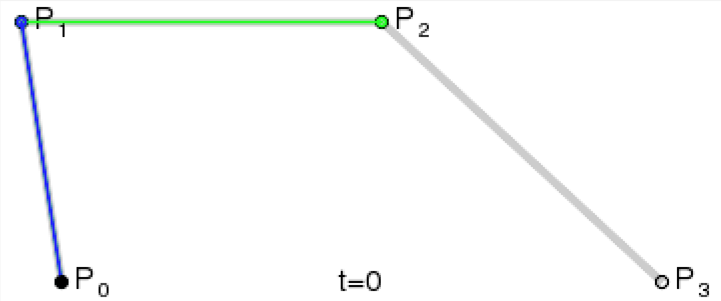




Path methods

bezierCurveTo(cp1X,cp1Y, cp2X,cp2Y, x,y)

- ○ Initial point – P0 – is the current "pencil" position
- ○ **cp1X**, **cp1Y**: control point 1 - P1 – coordinates
- ○ **cp2X**, **cp2Y**: control point 2 – P2 – coordinates
- ○ **x**, **y**: final point – P3 - coordinates



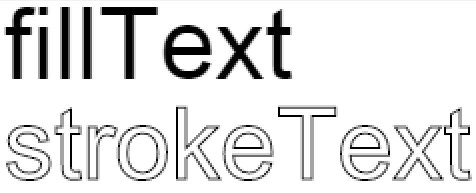
- Interact with the example here

Text

- Methods to write something in Canvas:

  fillText(text,x,y) or **strokeText(text,x,y)**

  - **text**: string to be written
  - **x**, **y**: start position of the text (first character)

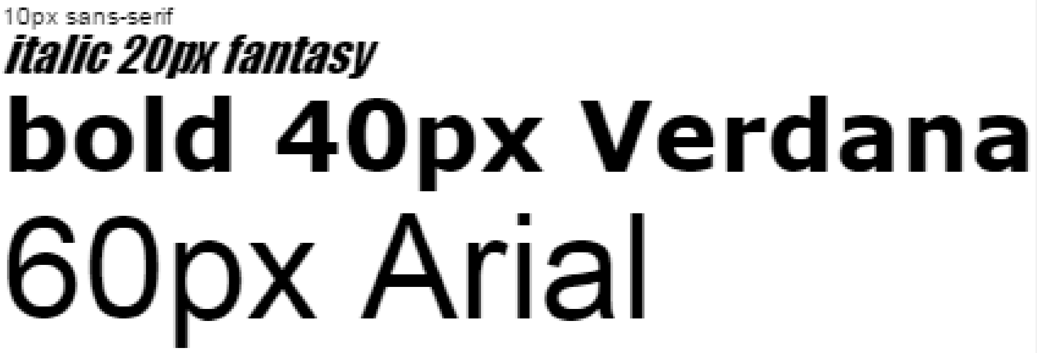


fillText
strokeText

- Some important text properties

  - **font**: default value - "10px sans-serif"

```
ctx.fillText(ctx.font, 10, 20);
ctx.font = 'italic 20px fantasy';
ctx.fillText(ctx.font, 10, 40);
ctx.font = 'bold 40px Verdana';
ctx.fillText(ctx.font, 10, 80);
ctx.font = '60px Arial';
ctx.fillText(ctx.font, 10, 140);
```

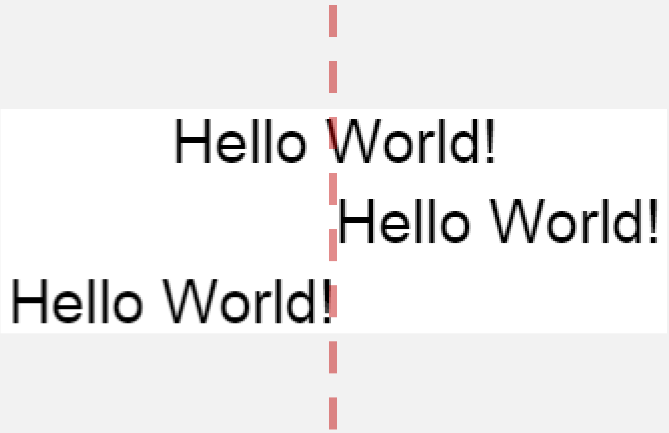


10px sans-serif
italic 20px fantasy
bold 40px Verdana
60px Arial

Text

- Some important text properties
  - `textAlign`: horizontal alignment - default value “left”

```
ctx.textAlign = 'center';
ctx.fillText("Hello World!", 300, 40);

ctx.textAlign = 'left';
ctx.fillText("Hello World!", 300, 60);

ctx.textAlign = 'right';
ctx.fillText("Hello World!", 300, 80);
```

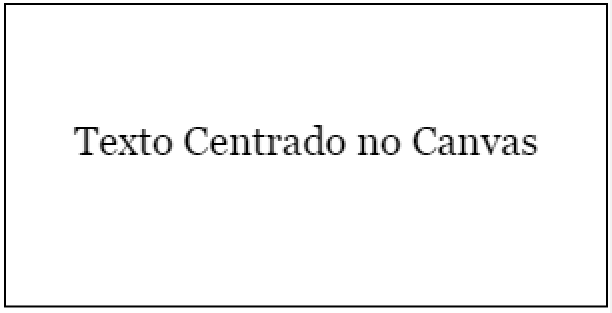


x = 300px

Text

- Some important text properties
  - **textBaseline**: vertical alignment - default value “alphabetic”

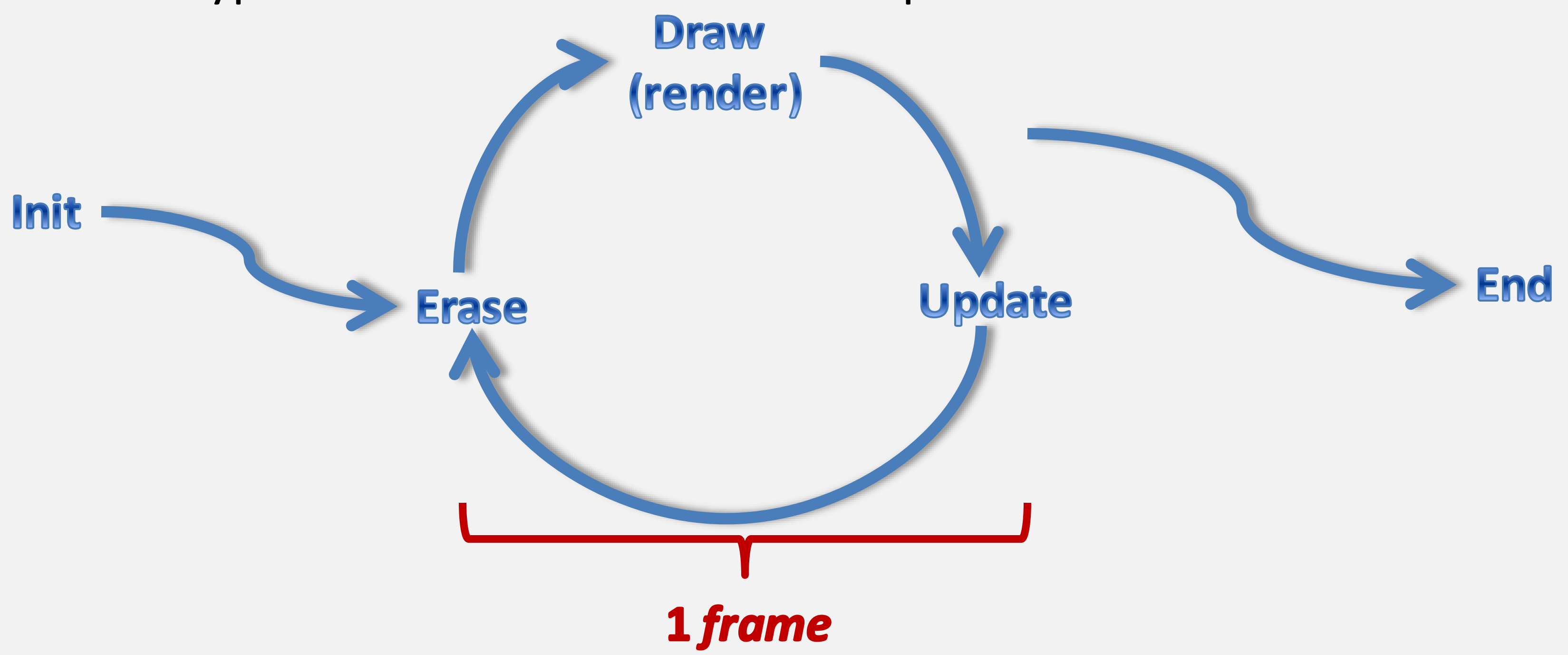
```
ctx.font = '20px Georgia';
var text = "Texto Centrado no Canvas";
ctx.textAlign = "center"; //alinhamento horizontal ao centro
ctx.textBaseline = "middle"; //alinhamento vertical ao centro
ctx.fillText (text, canvas.width/2, canvas.height/2);
```



Texto Centrado no Canvas

Animation cycle

- Creating a canvas **animation** is no more than drawing multiple times the same object (or objects)
  - just like frames in a video
- A typical animation works in a loop:



Animation cycle

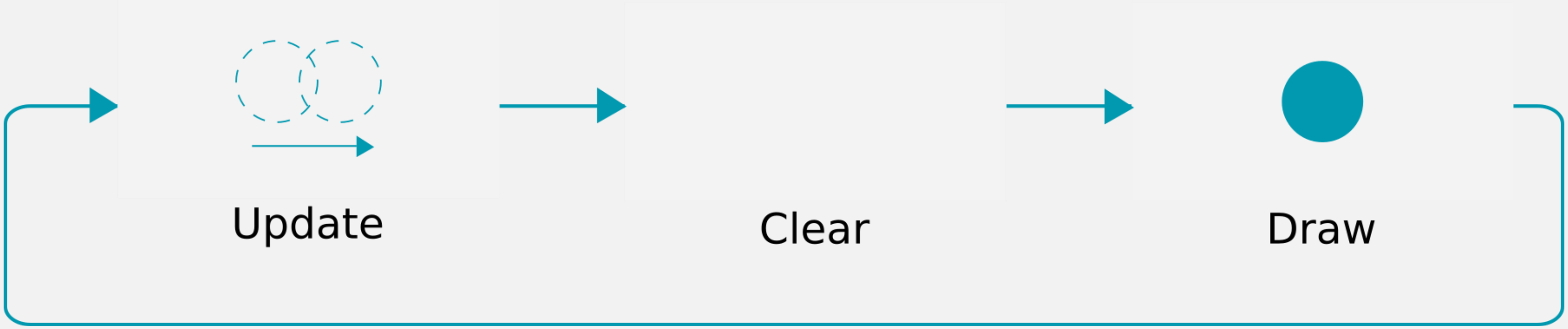
- How to control the animation loop?

`window.setInterval(callback, delay)`

timer that **repeats** `callback` function each delay milliseconds

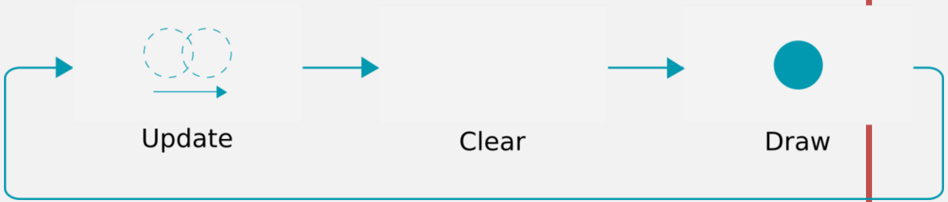
`window.requestAnimationFrame(callback)`

calls `callback` function whenever possible (**only once**)



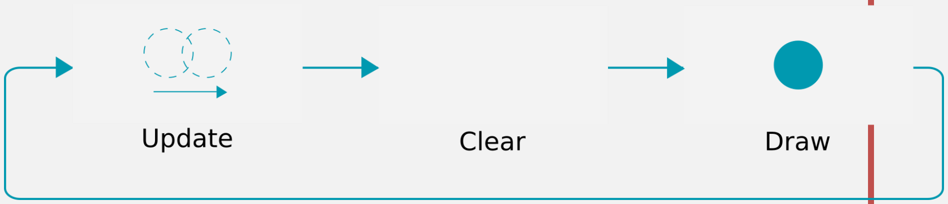
Animation cycle – using timers

```
const canvas = document.querySelector("#canvas");
const ctx = canvas.getContext("2d");

// animation control
let running = true; let timer;

function render(){
     ctx.fillRect(0, 0, canvas.width, canvas.height); // clear Canvas…
     // draw something…
     // update objects in drawing…

     if (!running)
          window.clearInterval(timer); // stop requesting new frames
}

window.onload = function(){
     timer = window.setInterval(render, 10);
};
```



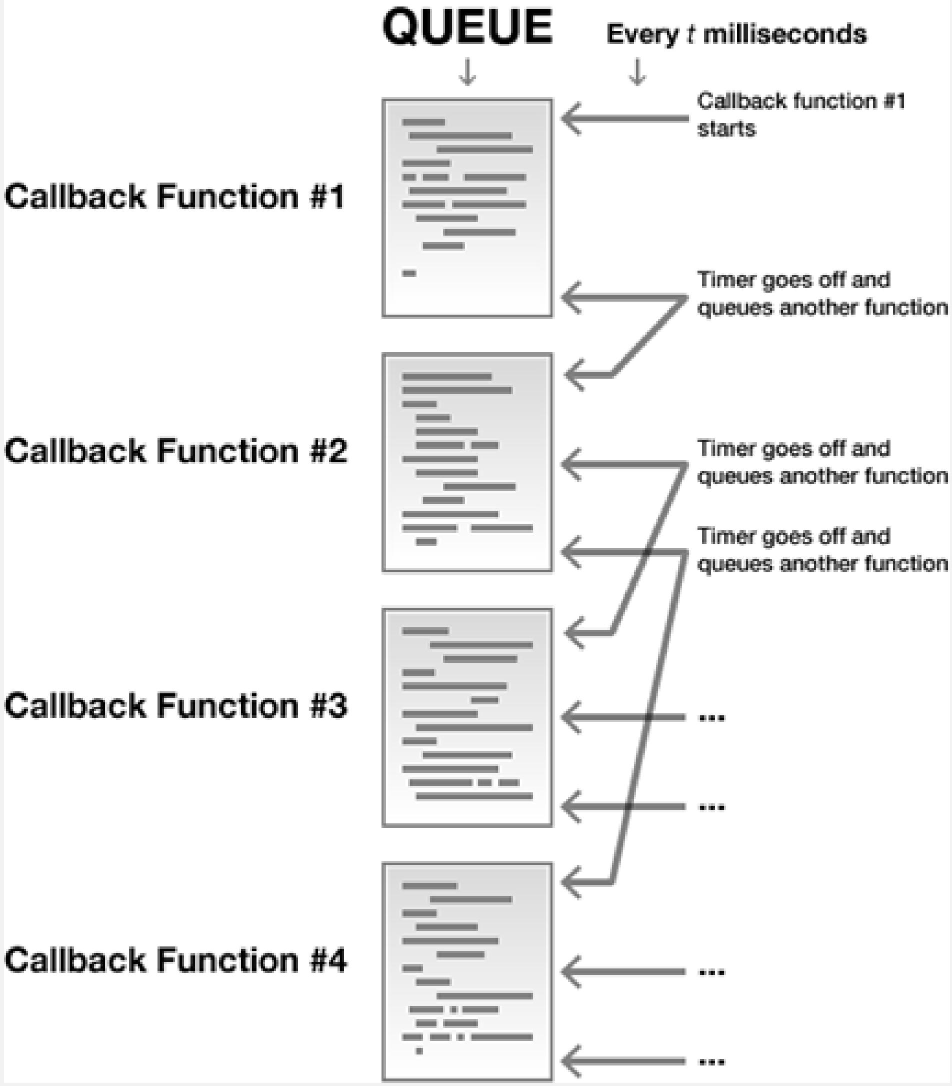
Animation cycle – by frames

```
const canvas = document.querySelector("#canvas");
const ctx = canvas.getContext("2d");

// animation control
let running = true;

function render(){
    ctx.fillRect(0,0, canvas.width, canvas.height); // clear Canvas…
    // draw something…
    // update objects in drawing…

    if (running)
        window.requestAnimationFrame(render); // keep requesting new frames
}

window.onload = render();
```

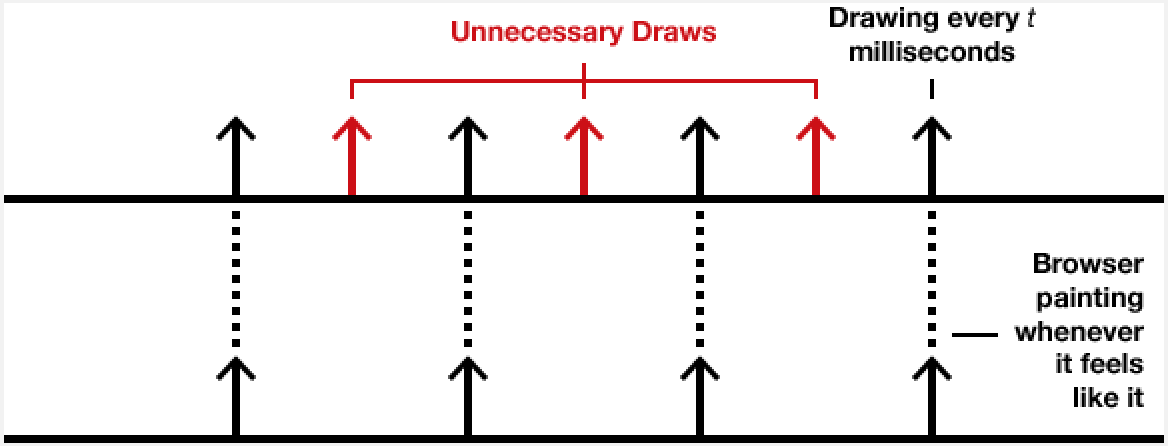


Animation cycle

Better performance with **requestAnimationFrame:**
If your callback functions take longer than your timers, enqueuing of multiple callback functions can choke up the browser


QUEUE
Every t milliseconds
Callback function #1 starts
Callback Function #1
Timer goes off and queues another function
Callback Function #2
Timer goes off and queues another function
Timer goes off and queues another function
Callback Function #3
...
...
Callback Function #4
...
...

Animation cycle

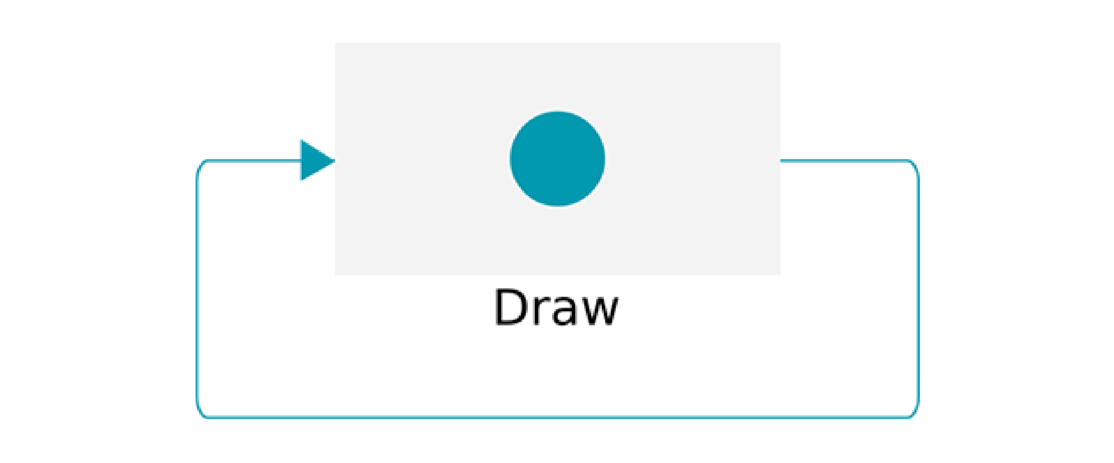

Unnecessary Draws
Drawing every *t* milliseconds
Browser painting whenever it feels like it

Better performance with requestAnimationFrame:
Unnecessary drawing, as some frames are drawn before the display refresh rate is ready to paint the animation outcome and are therefore just discarded. Skipped frames can lead to higher CPU usage and battery consumption, and sometimes even choppy animations

Animation: Knight Rider example

Download example file from Moodle, and observe the animation:

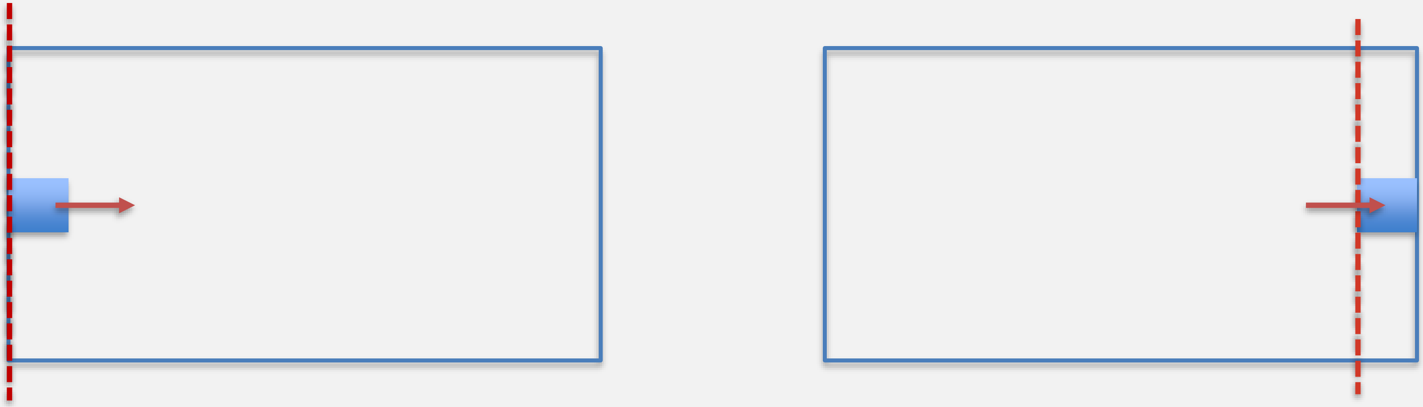
- When running this code, **`render()`** is now called with a high framerate
- The square is redrawn roughly 60 times per second, depending on the device you run the loop on
- Every frame, a new random color is picked
- This makes it easier to see the animation loop is actually working (but isn't very pleasant to look at)



Animation: Knight Rider example

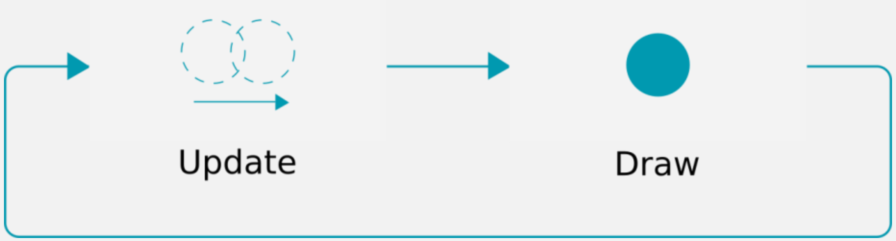
Alter the code so that you complete the following steps:

1. **Decrease** the animation framerate to 10 frames per second
2. **Update** the position of the square, each time it is draw: add 50 pixels to its X-coordinate



`minX = 0`

`maxX = canvas.width – square.width`

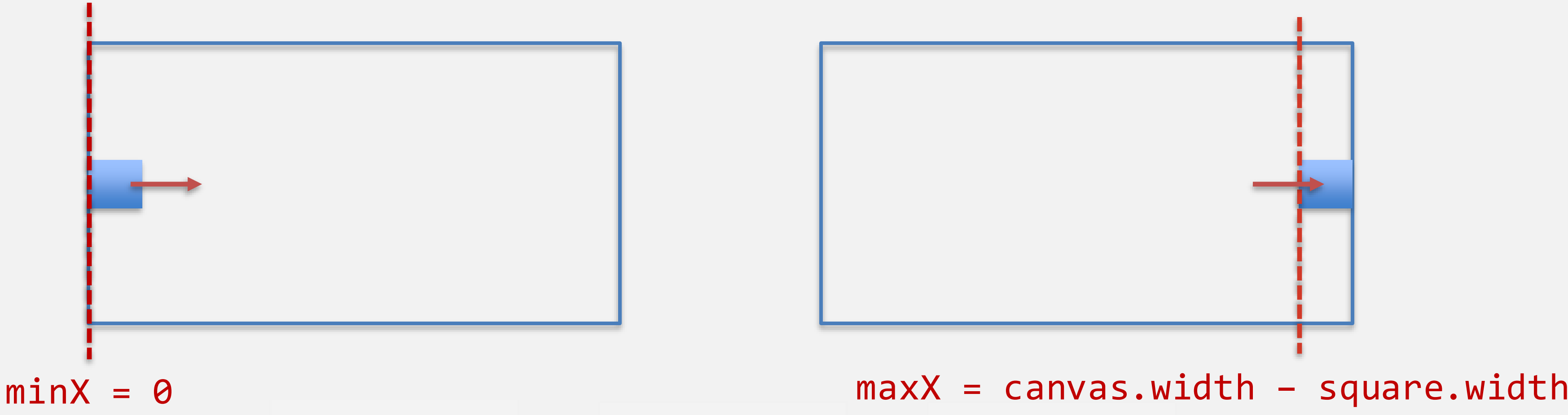


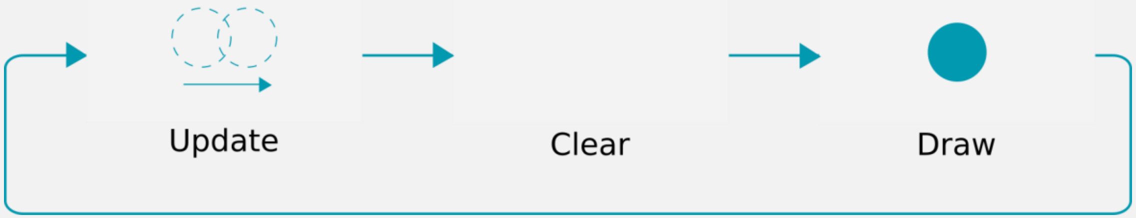
Animation: Knight Rider example

Alter the code so that you complete the following steps:

3. **Clear the Canvas** on each draw call, but by painting it with a rectangle of color `"rgba(51,51,51,0.3)"`

4. Make the square **bounce** between Canvas limits

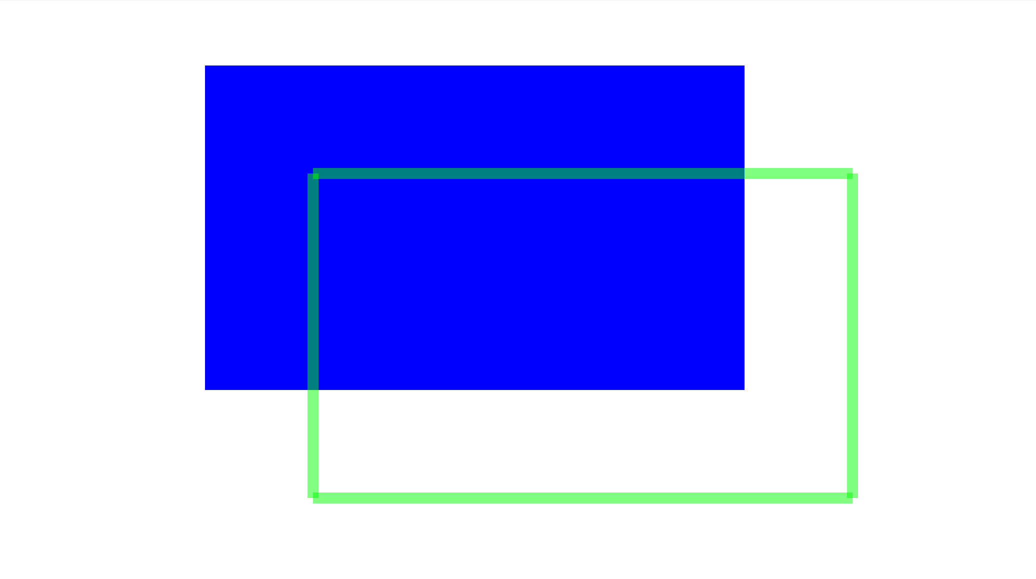




Try yourself...

Download from Moodle file `Ex01_lines.html` and code the solutions in where you find the comments //TODO

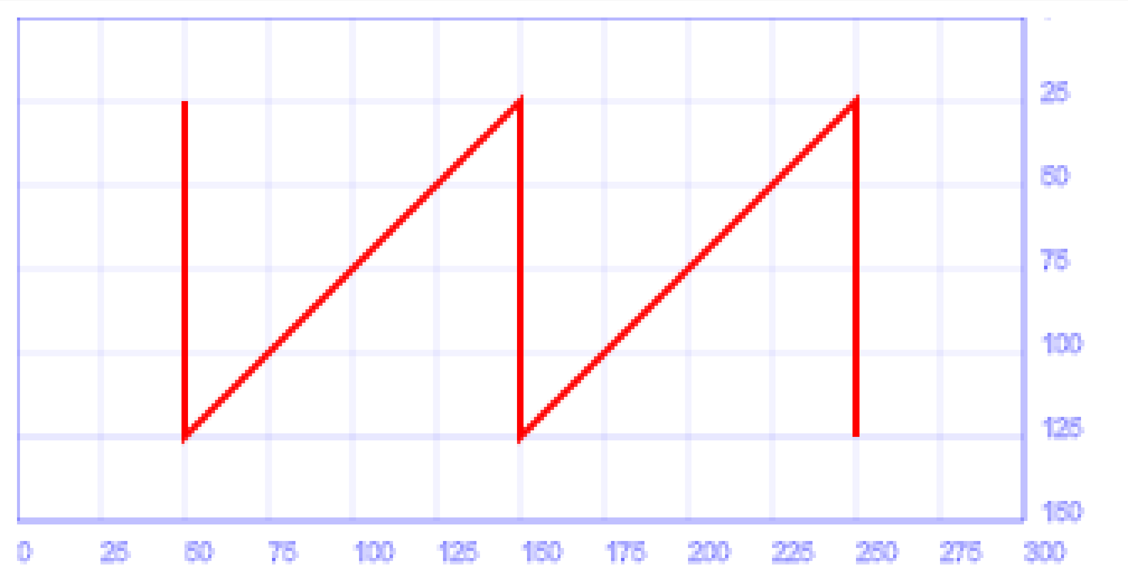
1. Using only lines, draw the following rectangles
   - What happens if you forget to create a new path, before drawing the green rectangle?



Try yourself…

Download from Moodle file `Ex01_lines.html` and code the solutions in where you find the comments //TODO

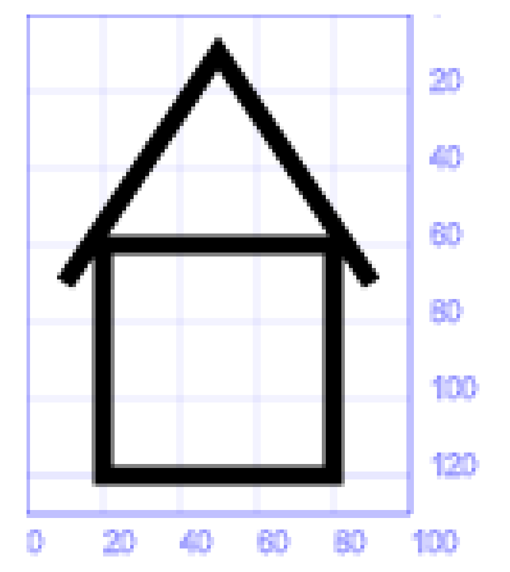
2. Draw the following shape (ignore the grid in image)
   - What happens if you draw the path using `fill()` instead of `stroke()`?



Try yourself...

Download from Moodle file `Ex01_lines.html` and code the solutions in where you find the comments //TODO

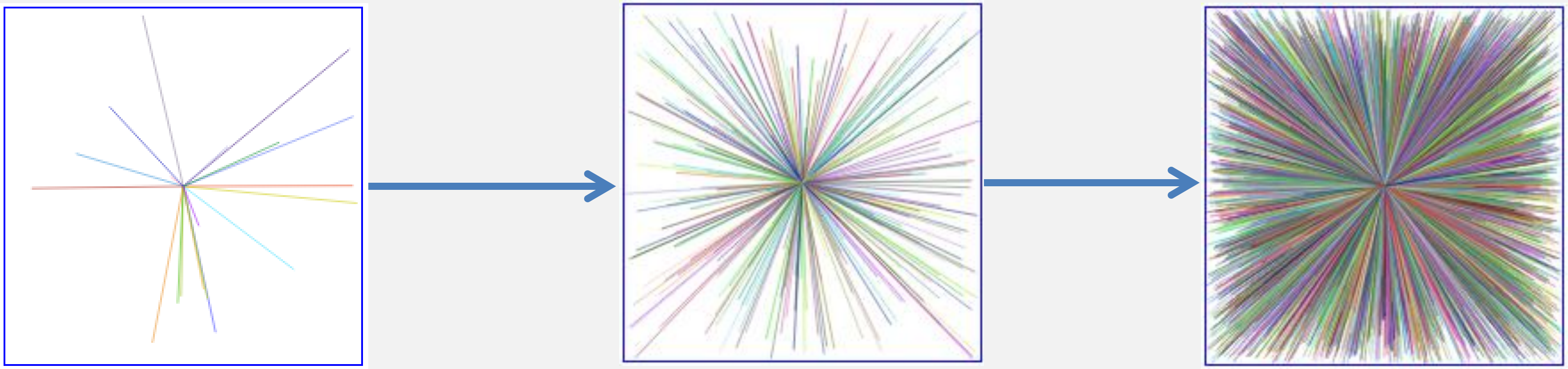
3. Draw the following shape (ignore the grid in image)
   - Use `rect()` to draw the house and two lines for the roof
   - Use property `linewidth` (default = 1) to increase the line contours width



Try yourself...

Download from Moodle file `Ex01_lines.html` and code the solutions in where you find the comments //TODO

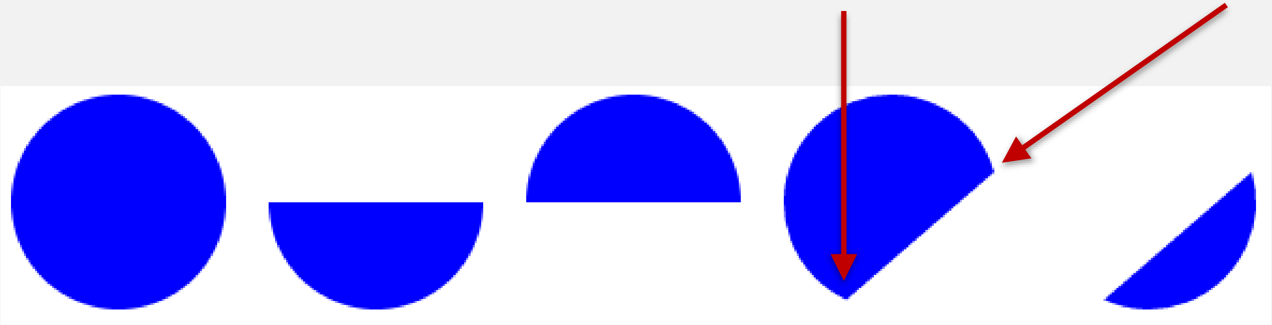
4. Make the following line animation
   - As fast as possible, draw one line per frame
   - Each line starts in the middle of the Canvas, and ends in a random point (inside the Canvas)
   - Each line has its own random color



Try yourself...

Download from Moodle file `Ex02_arcs&curves.html` and code the solutions in where you find the comments //TODO

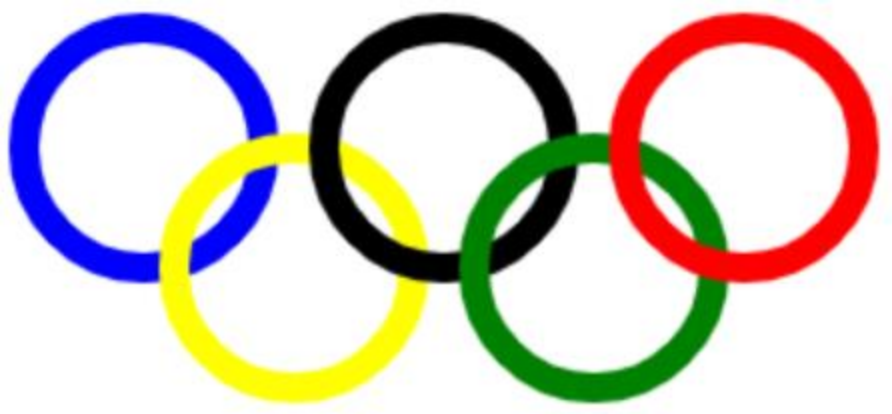
1. Draw the following figure using arcs
   - For the last two arcs, use angles `3*Math.PI/5` and `9*Math.PI/5`



Try yourself…

Download from Moodle file `Ex02_arcs&curves.html` and code the solutions in where you find the comments //TODO

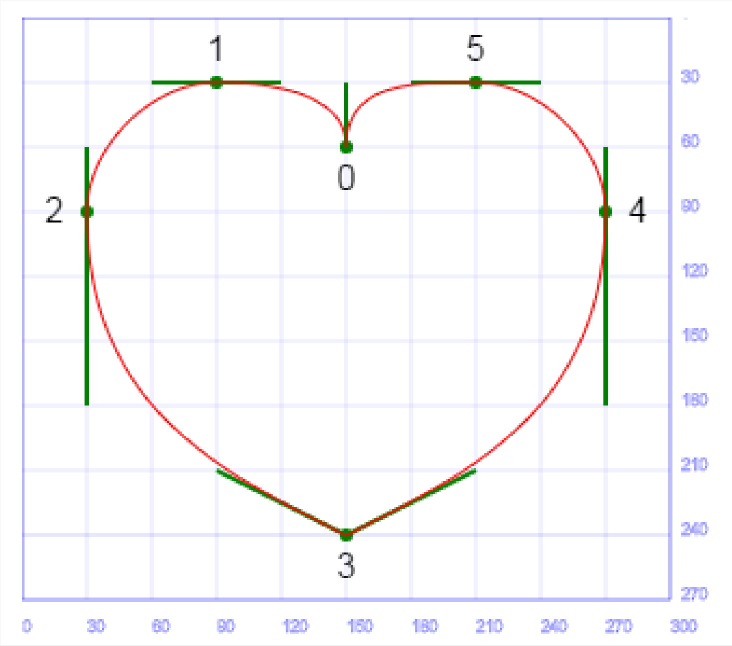
2. Draw the Olympic rings
   - Increase the line width using property `lineWidth`
   - All arcs have 40 pixels radii



Try yourself…

Download from Moodle file `Ex02_arcs&curves.html` and code the solutions in where you find the comments //TODO

3. Draw a heart, using 6 Bezier curves
   - Check image for help and the provided code as a start for your solution



```
//start at point 0 – little green circle
ctx.moveTo(150,60);

//1st curve: end at point 1
ctx.bezierCurveTo(150,30,120,30,90,30);

//draw others
//TODO

//stroke heart with red line
ctx.strokeStyle = "red";
ctx.stroke();
```

point 1

Green line end at point 0

Green line end at point 1

Try yourself...

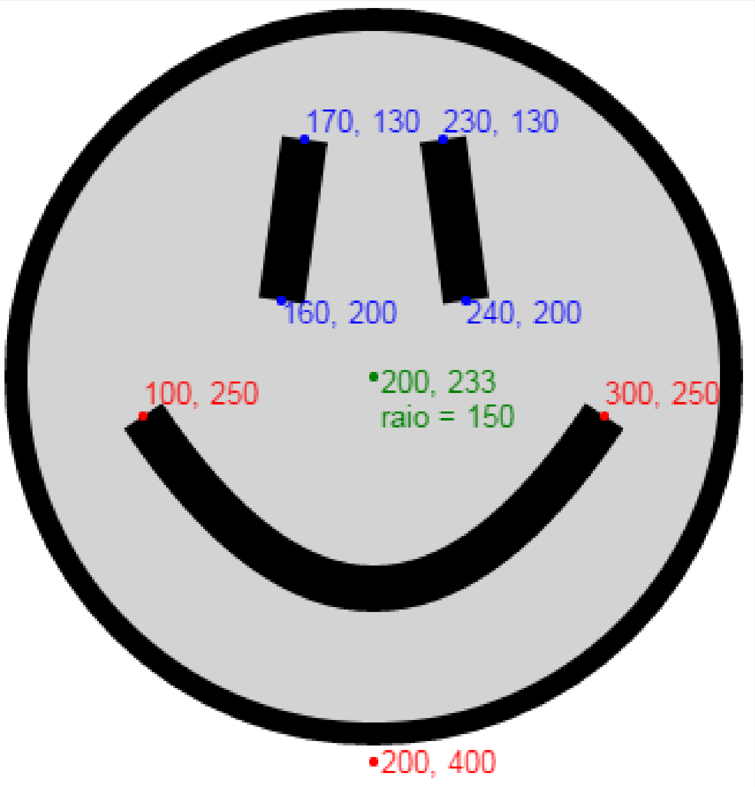
Download from Moodle file `Ex02_arcs&curves.html` and code the solutions in where you find the comments //TODO

4. Let's smile!
    - ○ Use the colored points to guess the shape and use them just for reference (do not draw them!)

      Face: color "lightgrey"

      Eyes

      Mouth
    - ○ All lines have a width of 20 pixels
    - ○ Can you **animate** your smiley, making it shift between a happy and a sad smiley?


170, 130
230, 130
160, 200
240, 200
100, 250
200, 233
raio = 150
300, 250
200, 400