**matter.min.js**.

It is a library file that allows our JavaScript code to use 2D physics engine properties.

When we use this library in our project folder, we’ll also need to add it to the **index.html** file so that we can use its functionalities in our code.

We’ll add the physics engine library called **matter.min.js.** in **index.html**.

In order to make use of this library, we need to import the necessary modules in our **sketch.js.** We import these modules from the **matter.min.js** library.

To use the physics engine, we will use three objects created in the **Matter.min.js library -> World, Engine and Bodies**.

Since these objects are a part of the **Matter.min.js** library, they are referenced as **Matter.World, Matter.Engine and Matter.Bodies.**

Let's rename them as **World, Engine**, and **Bodies** so that it is easy to write and read our code. This is called **namespacing** in programming.

Here **const** is just like var - except you cannot change the value stored in a **const** anywhere in your program.

● The engine is used to create the physics engine.

● World is used to create the physical world and add objects to it.

● Bodies are used to create the physical objects which inhabit the world.

As soon as you create a physics engine, a new world is created using the

physics engine.

Remember, this is an artificial world, where we can add different bodies and these bodies will have physical properties such as weight, velocity, position, etc.

This is very similar to the world we live in, such as you are in the world, and you have weight, height, and all the other such properties.

If someone bumps into you, you will move, if you jump you will come back down. All these rules are applied to all the objects, people, and animals around you.

These are called bodies and all bodies follow certain rules of the world.

Create a **canvas**, **engine**, and a **world** inside the function **setup()** in the **sketch.js** file.

we have a **Bodies.circle()** function that will help us to create the circle body. And to add the body to the world we’ll use the **World.add()** and pass the **world** and **ball** to it.

This is very important to understand that we can create multiple worlds as well.

Remember the example, in our solar system, where we have different planets and on each planet, we have different sets of rules for example on mercury there is no air, so there will be no air friction. The force of gravity is very high on Jupiter. On mars, the atmosphere is very thin.

That is why when we add the body to the world we need to specify to which world we are adding, since we have created a single world in our code, we will add our ball to that world itself.

Create a ball body using the **Bodies.circle()** function and add the **ball** body to the **world**.

We can also add physics properties to the ball. To add the properties, we’ll create a variable called **ball\_options**.

This object will store properties such as **restitution** and **frictionAir**.

**isStatic:**

If this is set to **true** the body will remain stationary else it can move, by default, this is set to **false**.

In the physics engine, we don't have to write code to specify the collision between bodies, all the bodies can collide with each other and have an effect on the other body, like when a ball collides with the ground it moves in

an upwards direction.

**Restitution:**

**Restitution** means how bouncy the ball is, the more the restitution the more bouncy the ball will be.

When we reduce the value of restitution, the body will be less bouncy.

Whereas **frictionAir** is the friction due to the air, more the friction slower the ball will move in the world.

We can have more properties like **weight, stiffness, drag**, etc., but we will explore them in the future classes.

After defining these properties, we pass this object to the **Bodies.circle()** function.

We have created the ball, but we still can’t see it on the screen because Physics Engine bodies do not get displayed on their own, it requires JavaScript objects or images to be placed on it. To display this ball, we will use an **ellipse()** function in the **draw()** function.

In this function, we’ll pass the **x** position, **y** position, and **radius** of the ball created using **Bodies.circle()**.

We’ll also need to continuously update the physics engine with the new changes. It is a necessary step as the function **draw()** is running every frame, so we need to update the engine to reflect changes as well as every frame.

To do so we’ll write **Engine.update(engine)**.

The x & y position is referred as **ball.position.x** and **ball.position.y draw()** function in **sketch.js**.

So thus the steps are:

1. **create Engine** -once => var engine =Engine .create();

2. **connect engine to world** -once => var world= engine.world

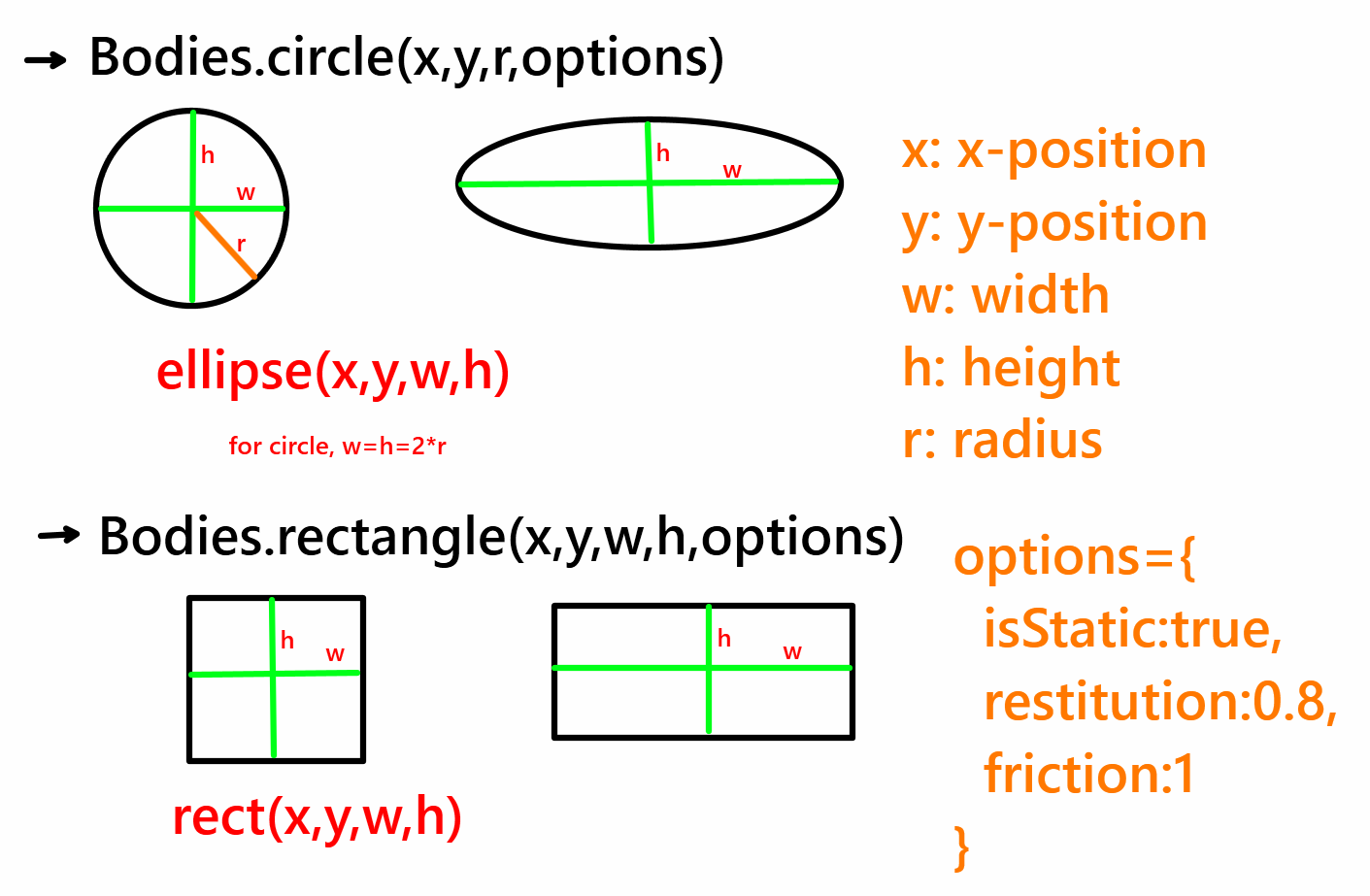
eg/ score =score+1

3. **create Bodies** -once => var ball = Bodies.circle(x,y,r,options);

4. **add bodies to world** -once => World.add(world, ball);

5. **draw the bodies** -always/ again ellipse(x,y,2\*r,2\*r)

6. start engine/ update engine -always/ again Engine .update(engine)



**Matter.Body.rotate()**

This function takes two parameters:

1) The body we want to rotate.

2) The angle by which we want it to rotate

Every time the **draw()** function is being called, the new positions that we have passed to the body revert to the old positions.

To solve this issue we use the **push()**, **pop()** and **translate()** functions.

**push()** function captures the new setting.

**pop()** function reverts to the old setting.

**translate()** function will translate(move) the coordinates of the canvas. Normally **0**,**0** is set in the top left corner of the canvas.

But with the **translate()** function, we will move it to the position of the body.

We are doing this because the rotate function rotates the shape around only the origin, if we don't translate the origin to the position of the body then the body will rotate around the top left corner, it will look like it is orbiting that

point.

But we want to rotate the body on its center point, which is why we need to translate the whole coordinate system to the center of the body.

But we want this setting only for the body, not for other bodies that is why we keep these functions in between a push() and pop() function.

So first we’ll use the **push()** to capture the new settings.

Then we’ll use the **translate()** function to pass it to the **x** and **y** position of the body.

Then using the **rotate()** function we’ll give it the angle of rotation.

Finally, use the **pop()** function to revert to the old settings.

We would also want to increase the angle of rotation by **0.1** degrees so that when the draw function runs we can see the body rotating.

So we’ll also write **angle +=0.1**