mdn web docs_

Adding features to our bouncing balls demo

In this assessment, you are expected to use the bouncing balls demo from the previous article as a starting point, and add some new and interesting features to it.

Prerequisites:	Before attempting this assessment you should have already worked through all the articles in this module.
Objective:	To test comprehension of JavaScript objects and object-oriented constructs

Starting point

To get this assessment started, make a local copy of index-finished.html, style.css, and main-finished.js from our last article in a new directory in your local computer.

Alternatively, you could use an online editor such as CodePen, JSFiddle, or Glitch. You could paste the HTML, CSS and JavaScript into one of these online editors. If the online editor you are using doesn't have a separate JavaScript panel, feel free to put it inline in a `<script>` element inside the HTML page.

Note: If you get stuck, you can reach out to us in one of our communication channels.

Hints and tips

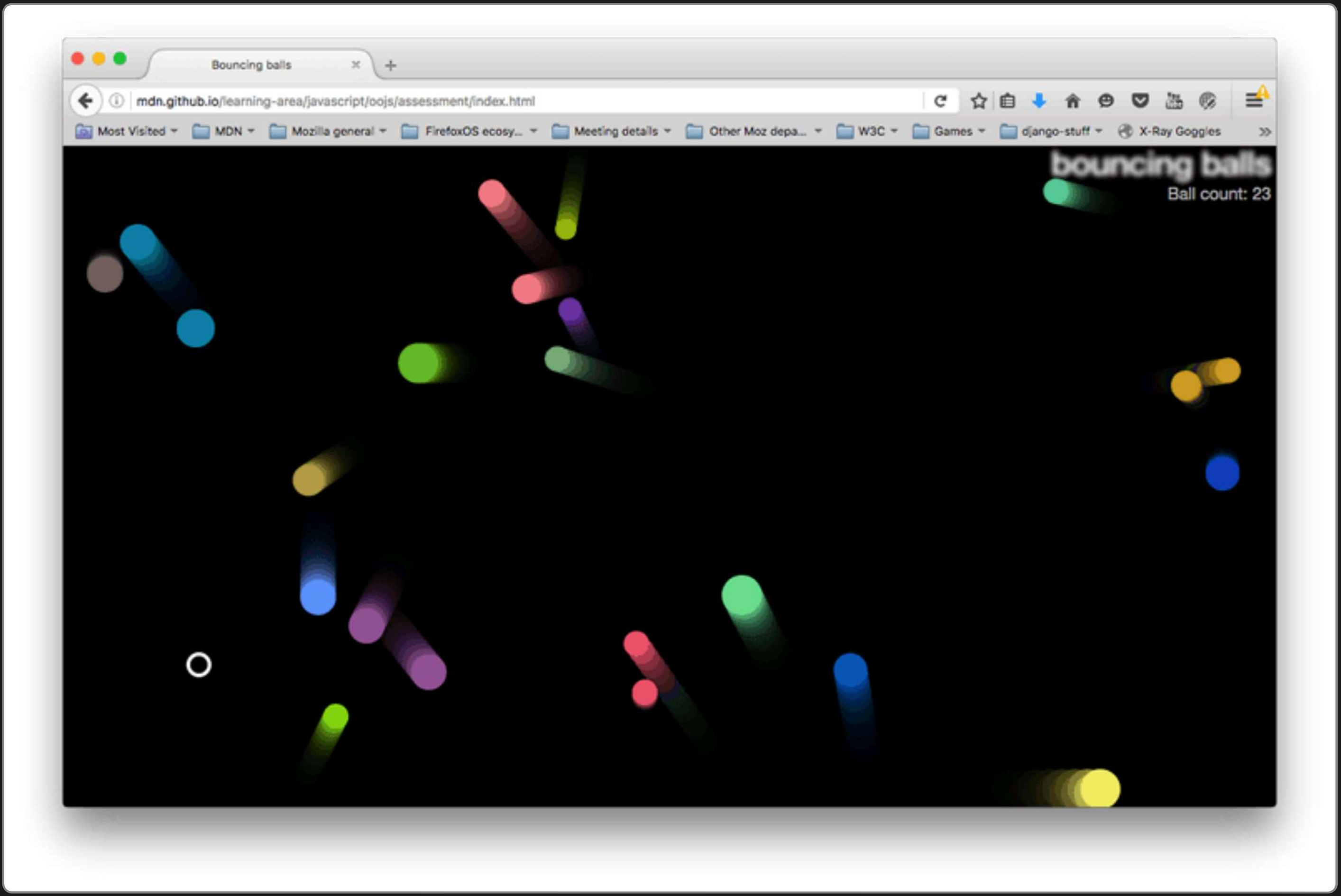
A couple of pointers before you get started.

- This assessment is quite challenging. Read the whole assessment before you start coding, and take each step slowly and carefully.
- It might be a good idea to save a separate copy of the demo after you get each stage working, so you can refer back to it if you find yourself in trouble later on.

Project brief

Our bouncy ball demo is fun, but now we want to make it a little bit more interactive by adding a user-controlled evil circle, which will eat the balls if it catches them. We also want to test your object-building skills by creating a generic `Shape()` object that our balls and evil circle can inherit from. Finally, we want to add a score counter to track the number of balls left to capture.

The following screenshot gives you an idea of what the finished program should look like:



To give you more of an idea, have a look at the finished example (no peeking at the source code!)

Steps to complete

The following sections describe what you need to do.

Create a Shape class

First of all, create a new `Shape` class. This has only a constructor. The `Shape` constructor should define the `x`, `y`, `velX`, and `velY` properties in the same way as the `Ball()` constructor did originally, but not the `color` and `size` properties.

The `Ball` class should be made to derive from `Shape` using `extends`. The constructor for `Ball` should:

- take the same arguments as before: `x`, `y`, `velX`, `velY`, `size`, and `color`
- call the `Shape` constructor using `super()`, passing in the `x`, `y`, `velX`, and `velY` arguments
- initialize its own `color` and `size` properties from the parameters it is given.

The `Ball` constructor should define a new property called `exists`, which is used to track whether the balls exist in the program (have not been eaten by the evil circle). This should be a boolean (`true`/`false`), initialized to `true` in the constructor.

The `collisionDetect()` method of the `Ball` class needs a small update. A ball needs to be considered for collision detection only if the `exists` property is `true`. So, replace the existing `collisionDetect()` code with the following code:

JS

```
collisionDetect() {
  for (const ball of balls) {
    if (!(this === ball) && ball.exists) {
      const dx = this.x - ball.x;
      const dy = this.y - ball.y;
      const distance = Math.sqrt(dx * dx + dy * dy);

      if (distance < this.size + ball.size) {
        ball.color = this.color = randomRGB();
      }
    }
  }
}
```

As discussed above, the only addition is to check if the ball exists — by using `ball.exists` in the `if` conditional.

The ball `draw()` and `update()` method definitions should be able to stay exactly the same as they were before.

At this point, try reloading the code — it should work just the same as it did before, with our redesigned objects.

Defining EvilCircle

Now it's time to meet the bad guy — the `EvilCircle()`! Our game is only going to involve one evil circle, but we are still going to define it using a constructor that inherits from `Shape()`, to give you some practice. You might want to add another circle to the app later on that can be controlled by another player, or have several computer-controlled evil circles. You're probably not going to take over the world with a single evil circle, but it will do for this assessment.

Create a definition for an `EvilCircle` class. It should inherit from `Shape` using `extends`.

EvilCircle constructor

The constructor for `EvilCircle` should:

- be passed just `x`, `y` arguments
- pass the `x`, `y` arguments up to the `Shape` superclass along with values for `velX` and `velY` hardcoded to 20. You should do this with code like `super(x, y, 20, 20);`
- set `color` to `white` and `size` to `10`.

Finally, the constructor should set up the code enabling the user to move the evil circle around the screen:

JS

```js
window.addEventListener("keydown", (e) => {
  switch (e.key) {
    case "a":
      this.x -= this.velX;
      break;
    case "d":
      this.x += this.velX;
      break;
    case "w":
      this.y -= this.velY;
      break;
    case "s":
      this.y += this.velY;
      break;
  }
});
```

This adds a `keydown` event listener to the `window` object so that when a key is pressed, the event object's `key` property is consulted to see which key is pressed. If it is one of the four specified keys, then the evil circle will move left/right/up/down.

Defining methods for EvilCircle

The `EvilCircle` class should have three methods, as described below.

draw()

This method has the same purpose as the `draw()` method for `Ball`: it draws the object instance on the canvas. The `draw()` method for `EvilCircle` will work in a very similar way, so you can start by copying the `draw()` method for `Ball`. You should then make the following changes:

- We want the evil circle to not be filled in, but rather just have an outer line (stroke). You can achieve this by updating `fillStyle` and `fill()` to `strokeStyle` and `stroke()` respectively.

- We also want to make the stroke a bit thicker, so you can see the evil circle a bit more easily. This can be achieved by setting a value for `lineWidth` somewhere after the `beginPath()` call (3 will do).

checkBounds()

This method will do the same thing as the first part of the `update()` method for `Ball` — look to see whether the evil circle is going to go off the edge of the screen, and stop it from doing so. Again, you can mostly just copy the `update()` method for `Ball`, but there are a few changes you should make:

- Get rid of the last two lines — we don't want to automatically update the evil circle's position on every frame, because we will be moving it in some other way, as you'll see below.
- Inside the `if ()` statements, if the tests return true we don't want to update `velX`/`velY`; we want to instead change the value of `x`/`y` so the evil circle is bounced back onto the screen slightly. Adding or subtracting (as appropriate) the evil circle's `size` property would make sense.

collisionDetect()

This method will act in a very similar way to the `collisionDetect()` method for `Ball` method, so you can use a copy of that as the basis of this new method. But there are a couple of differences:

- In the outer `if` statement, you no longer need to check whether the current ball in the iteration is the same as the ball that is doing the checking — because it is no longer a ball, it is the evil circle! Instead, you need to do a test to see if the ball being checked exists (with which property could you do this with?). If it doesn't exist, it has already been eaten by the evil circle, so there is no need to check it again.
- In the inner `if` statement, you no longer want to make the objects change color when a collision is detected — instead, you want to set any balls that collide with the evil circle to not exist any more (again, how do you think you'd do that?).

Bringing the evil circle into the program

Now we've defined the evil circle, we need to actually make it appear in our scene. To do this, you need to make some changes to the `loop()` function.

- First of all, create a new evil circle object instance (specifying the necessary parameters). You only need to do this once, not on every iteration of the loop.
- At the point where you loop through every ball and call the `draw()`, `update()`, and `collisionDetect()` functions for each one, make it so that these functions are only called if the current ball exists.
- Call the evil circle instance's `draw()`, `checkBounds()`, and `collisionDetect()` methods on every iteration of the loop.

Implementing the score counter

To implement the score counter, follow the following steps:

1. In your HTML file, add a `<p>` element just below the h1 element containing the text "Ball count: ".
2. In your CSS file, add the following rule at the bottom:

```css
p {
  position: absolute;
  margin: 0;
  top: 35px;
  right: 5px;
  color: #aaa;
}
```

3. In your JavaScript, make the following updates:
   - Create a variable that stores a reference to the paragraph.
   - Keep a count of the number of balls on screen in some way.
   - Increment the count and display the updated number of balls each time a ball is added to the scene.
   - Decrement the count and display the updated number of balls each time the evil circle eats a ball (causes it not to exist).

Help improve MDN

Was this page helpful to you?

Yes No

Learn how to contribute.

This page was last modified on Oct 23, 2023 by MDN contributors.