# Real Time Simulation

Class 22 – Prepared by Nicolas Bergeron

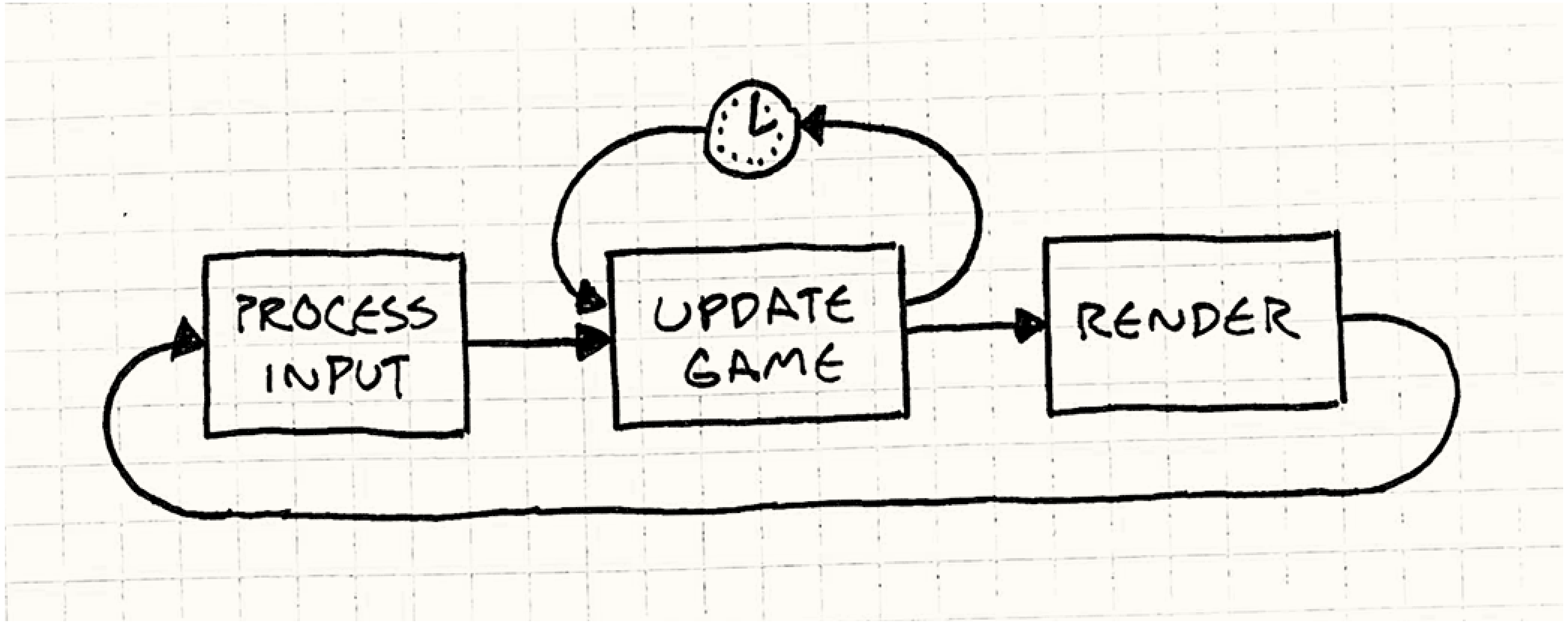
## Outline

- Real Time Simulation (Games)
- Main Loop
- Physics Euler Integration
- Setting up Scene for Real Time Simulation Game
    - Apply gravity to shape
- Dynamically adding nodes to your scene
- Exercise

## Real Time Simulation

- Real time simulation applications are applications where a Scene is composed of multiple objets, and the state of each object depends on the current time in the application
- Video Games are examples of real time simulation applications
- In a real-time simulations, the state of objects is updated at least 30 times per second, and each time is rendered on the screen. This process is called the Main Loop

## The Main Loop

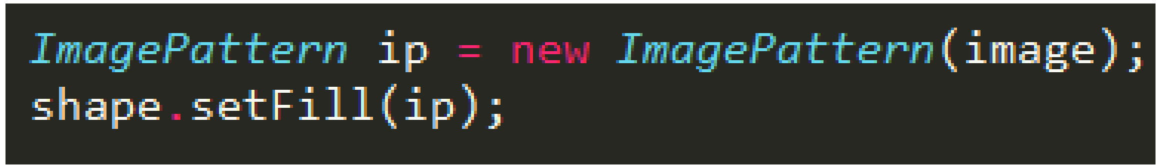


## Timing information

- In Java, the current time can be retrieved using System.nanoTime()
- The method returns an integer (long) representing the most precise timer on the system in nano second.
- We can measure time spent by substracting nano seconds from one call to another. To get the timing in seconds, we can divide the time interval by a billion (1000000000)
- To simulate a Main Loop, we can start an AnimationTimer. The handle method will automatically be called as often as possible. The amount of time between 2 calls of the handle() method will determine the frame duration.

# JavaFX Shapes for Simulation

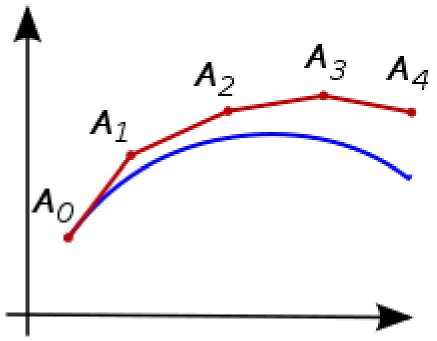
- JavaFX provides Shape objects that are convenient to use for simulation applications.
- Circle objects can be positioned on a Pane using the setCenterX() and setCenterY() methods. It can be resized using the setRadius() method.
- Rectangle objects can also be positioned with setX() and setY(). The position where they draw is at the upper left corner by default.
- For any shape (variable name), to set a background image (JavaFX Image class), you can use the code below:



```
ImagePattern ip = new ImagePattern(image);
shape.setFill(ip);
```

# Physics - Euler Integration

- Re-compute Acceleration, Velocities and Position once per time step
  - Apply gravity and other forces to acceleration
  - Apply acceleration to velocity
  - Apply velocity to positions

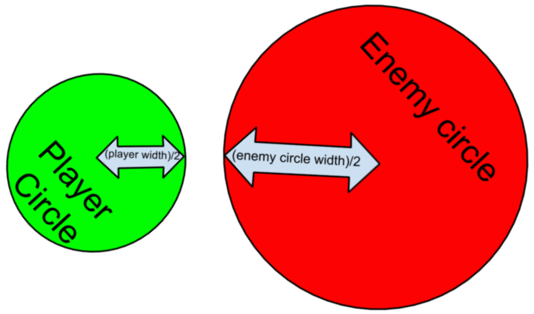


**Typical Problems**

Blue: Correct Positions
Red: Positions from Euler Integration

- Advantages with Euler Integration
  - Easy to implement
  - Fast to compute
- Problems with Euler Integration
  - Not Accurate, but good enough for simple movement
  - More robust integrations techniques (such as RK4) are required for more accurate physics simulation

# 2D Physics - Collisions

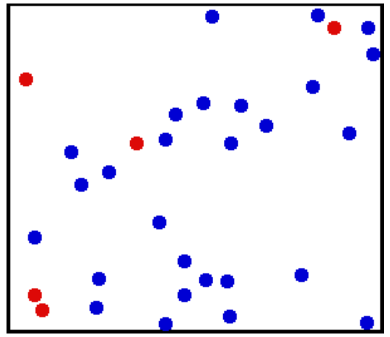


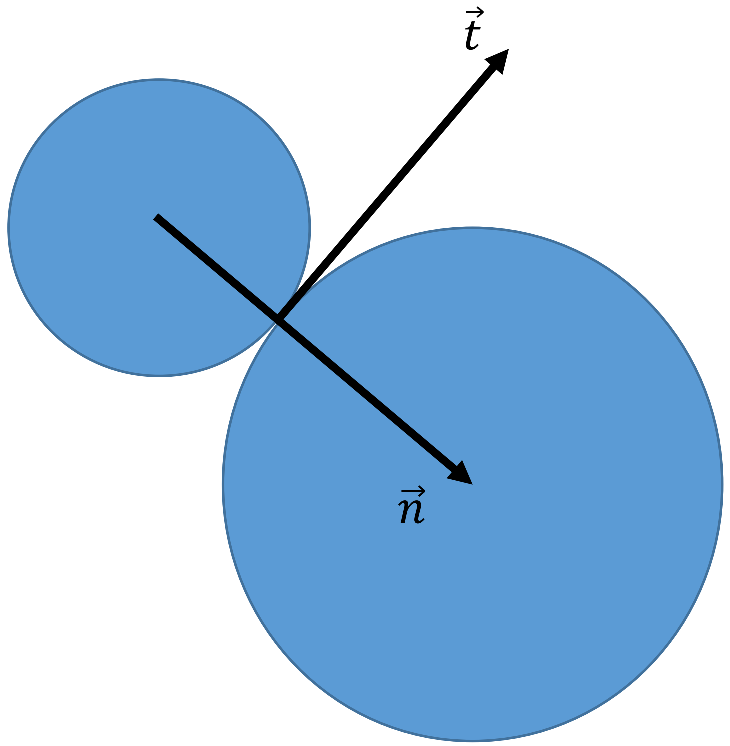
- **Projectiles at constant speed** (inherit PhysicsGameObject)
  - Instanciate a projectile at an initial position
  - Set the velocity vector towards a destination (v = destination – initial)
  - Set the magnitude of your vector to be the speed you need (normalize then multiply by speed)
  - You can set the acceleration to be 0 and ignore gravity for projectiles

- **Detecting collision between 2 circles of arbitrary position** (Vector2 $c_1$ and $c_2$) and radius (float $r_1$ and $r_2$)
  - Calculate distance between 2 centers ($c_1$ and $c_2$)

    $$d = \sqrt{(c_2.x - c_1.x)^2 + (c_2.y - c_1.y)^2}$$
  - if ($d < r_1 + r_2$), then the 2 circles collide
- You can get projectiles collision circle approximately matching the size of the sprite
- Every frame, detect collision between projectiles and targets (eg: enemies)

## 2D Physics – Circle Collision Response

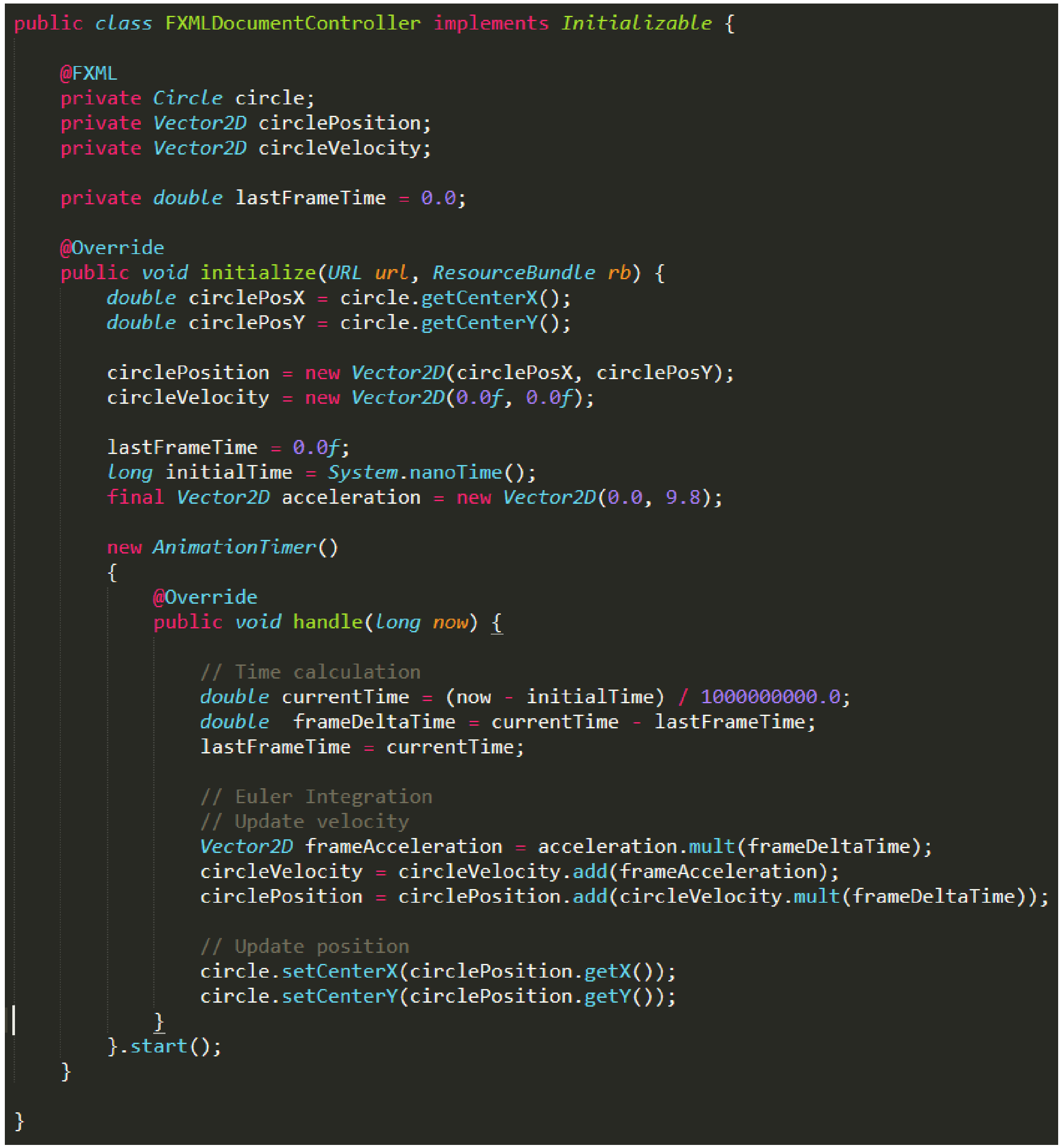




- 2 Circles
  - centered at $C_1$ and $C_2$
  - with velocities $\overrightarrow{v_1}$ and $\overrightarrow{v_2}$
- Define a normal and tangent vectors
  - $\vec{n} = C_2 - C_1$ (and normalize $\vec{n}$)
  - $\vec{t} = (-n_y, n_x)$ (perpendicular to $\vec{n}$)
- Decompose both velocities according to $\vec{n}$ and $\vec{t}$
  - $\overrightarrow{v_1}^t = \overrightarrow{v_1} \cdot \vec{t}$ $\quad \overrightarrow{v_1}^n = \overrightarrow{v_1} \cdot \vec{n}$
  - $\overrightarrow{v_2}^t = \overrightarrow{v_2} \cdot \vec{t}$ $\quad \overrightarrow{v_2}^n = \overrightarrow{v_2} \cdot \vec{n}$
- Velocities after collision $\overrightarrow{v_1}$'and $\overrightarrow{v_2}$'
  - $\overrightarrow{v_1}' = \overrightarrow{v_1}^t + \overrightarrow{v_1}^n - \overrightarrow{v_2}^n$
  - $\overrightarrow{v_2}' = \overrightarrow{v_2}^t + \overrightarrow{v_2}^n - \overrightarrow{v_1}^n$

# Scene Setup in Java FX

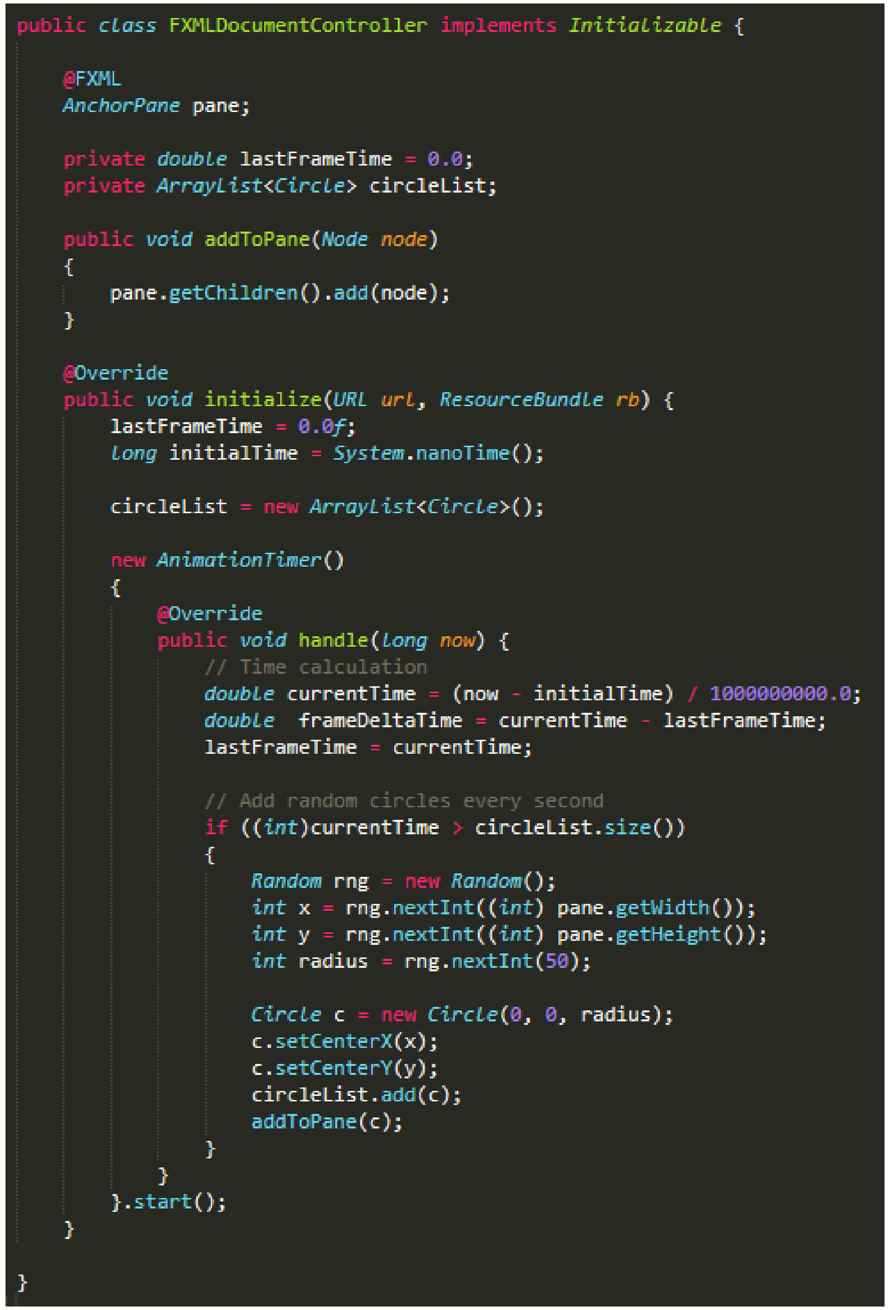
- Add a circle to your scene
- Update the circle every frame using an Animation Timer that you start in the initialization method
- Every frame (timer tick), you will update the position of the circle by taking into account the current velocity, updated by the acceleration.



```java
public class FXMLDocumentController implements Initializable {

    @FXML
    private Circle circle;
    private Vector2D circlePosition;
    private Vector2D circleVelocity;

    private double lastFrameTime = 0.0;

    @Override
    public void initialize(URL url, ResourceBundle rb) {
        double circlePosX = circle.getCenterX();
        double circlePosY = circle.getCenterY();

        circlePosition = new Vector2D(circlePosX, circlePosY);
        circleVelocity = new Vector2D(0.0f, 0.0f);

        lastFrameTime = 0.0f;
        long initialTime = System.nanoTime();
        final Vector2D acceleration = new Vector2D(0.0, 9.8);

        new AnimationTimer()
        {
            @Override
            public void handle(long now) {

                // Time calculation
                double currentTime = (now - initialTime) / 1000000000.0;
                double  frameDeltaTime = currentTime - lastFrameTime;
                lastFrameTime = currentTime;

                // Euler Integration
                // Update velocity
                Vector2D frameAcceleration = acceleration.mult(frameDeltaTime);
                circleVelocity = circleVelocity.add(frameAcceleration);
                circlePosition = circlePosition.add(circleVelocity.mult(frameDeltaTime));

                // Update position
                circle.setCenterX(circlePosition.getX());
                circle.setCenterY(circlePosition.getY());
            }
        }.start();
    }

}
```

# Dynamically adding Nodes to your Scene

- You can keep track of the root pane in your controller
- From the pane, you can add any node (shape, button, etc) to your scene
- In the example, we instantiate a new circle every second and keep track of them in a container (array list)



```java
public class FXMLDocumentController implements Initializable {

    @FXML
    AnchorPane pane;

    private double lastFrameTime = 0.0;
    private ArrayList<Circle> circleList;

    public void addToPane(Node node)
    {
        pane.getChildren().add(node);
    }

    @Override
    public void initialize(URL url, ResourceBundle rb) {
        lastFrameTime = 0.0f;
        long initialTime = System.nanoTime();

        circleList = new ArrayList<Circle>();

        new AnimationTimer()
        {
            @Override
            public void handle(long now) {
                // Time calculation
                double currentTime = (now - initialTime) / 1000000000.0;
                double  frameDeltaTime = currentTime - lastFrameTime;
                lastFrameTime = currentTime;

                // Add random circles every second
                if ((int)currentTime > circleList.size())
                {
                    Random rng = new Random();
                    int x = rng.nextInt((int) pane.getWidth());
                    int y = rng.nextInt((int) pane.getHeight());
                    int radius = rng.nextInt(50);

                    Circle c = new Circle(0, 0, radius);
                    c.setCenterX(x);
                    c.setCenterY(y);
                    circleList.add(c);
                    addToPane(c);
                }
            }
        }.start();
    }

}
```

## Exercise

Exercise 1

- From the first example, make the circle bounce by flipping the velocity “y” vector component when the circle reaches the bottom of the pane. You can also add a multiplier between 0 and 1 to make it eventually rest.

Also Setup your Assignment 3

Be prepared to demo!

Exercise 2

- From the second example, add gravity and bouncing to each circles spawned over time.
- Collisions:
  - Step 1: Make circles disappear when they collide
  - Step 2: Make start with random velocity vectors, and bounce off each other
- You could create a class GameObject containing a circle, acceleration, velocity and position Vector2D to simplify your code
  - Update method receives the duration of the frame (delta time)

## References

- JavaFX for Game Development https://gamedevelopment.tutsplus.com/tutorials/introduction-to-javafx-for-game-development--cms-23835