Exercise NMCGJ: Animated Graphics

Animation is a rapid display of a sequence of graphics or images which creates an illusion of movement. Animation is an important feature in game programming. In this exercise we will produce some animated graphics.

Animation in Java

Animation is a process that continues over a certain period of time, and this process is often terminated by an external (often user based) action. Unfortunately, this creates a difficult problem! Let us look at the following example. To create an animation we would like to have a code like

```
public class AnimationClass {

   boolean animated = true;

   public void playAnimation() {
      while (animated) {
         doAnimationStep();
      }
   }

   public void stopAnimation() {
      animated = false;
   }

   public void doAnimationStep() {
      // the code that takes care of an animation step
   }

}
```

The method `doAnimationStep` takes care of one step in our animation. Now, we would like to start this animation by calling the method `playAnimation` and to stop the animation by calling the method `stopAnimation`. This implies that an external action (for example, input by the user) can control this process.

However, this piece of code will never work! Once the while loop in `playAnimation` has started, it will occupy the CPU and it will never respond to external actions. This means that we get an infinite loop! The solution to solve this problem is working with so-called threads.

Threads

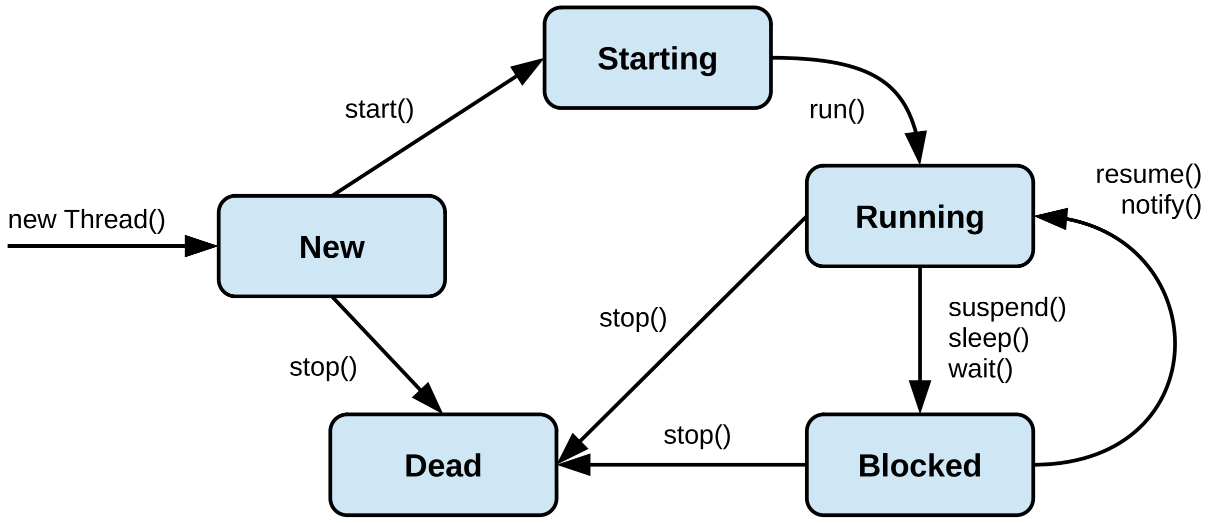
Java is a multi-threaded programming language, which means that a Java program can contain two or more parts (called threads) that run concurrently and each thread can handle a different task at the same time. Roughly speaking, the CPU will pop around and give each thread some of its time. Each thread has the impression of constantly having the CPU to itself, but actually the CPU's time is being sliced among all the threads (except when the program is running on multiple CPUs). In this way, one can also make optimal use of the available resources: if one thread is inactive, the other thread(s) can still continue to work.

Anything done in a Java program that runs continually and takes up a lot of processing time should run in its own thread. Animation is one of these things. One way to create a thread is by writing a class that inherits from the class `Thread`. The method `run` should be implemented to take care of the specific task. So, in the example above, we can change our code into:

```
public class AnimationClass extends Thread {

   public void run() {
      playAnimation();
   }

   // the other code...

}
```

To accomplish our animation, the `start` method should be called to start a new thread. This will automatically call the thread's `run` method, so activating all the animation processing inside the `run` method. This allows the animation to run on its own without interfering with any other parts of the program.

In addition, the class `Thread` offers functionality to let the thread sleep for a while (meaning the execution of the task is blocked for a while), or to run as a daemon thread (meaning it runs as a general service in the background). The typical life cycle of a thread looks like:



Concurrent programming is a quite complicated matter, and its full complexity is beyond the scope of this course.

The class AnimationGraphicsFrame

The class `AnimationGraphicsFrame` is prepared for making animated 2D graphics. It is a subclass of the class `GraphicsFrame`, so it provides all the functionality of this class (see the previous exercise *Drawing Fractal Patterns*). In addition, the following methods are available to run an animation.

playAnimation()	Plays the animation when not currently active or resumes the animation when paused. It has no effect if the animation is already running.
pauseAnimation()	Pauses the animation.
stopAnimation()	Terminates the animation.
isAnimationEnabled(): boolean	Indicates whether the animation is enabled or not.
isAnimationPaused(): boolean	Indicates whether the animation is paused or not.
setAnimationDelay(long millis)	Sets the delay time (in milliseconds) between each animation step.

Each animation is executed in a new thread. The first three methods are called, respectively, by the menu items Animation > Play, Animation > Pause, and Animation > Stop.

The class `AnimationGraphicsFrame` is an abstract class, and provides an animation environment. It requires the implementation of the following abstract methods:

animateInit()	Initializes a new animation (is called before the start of the animation).
animateNext()	Executes the next step in the animation.
animateFinal()	Finalizes the animation (is called after the end of the animation).

These methods have to be implemented by a subclass and define the specific animation.

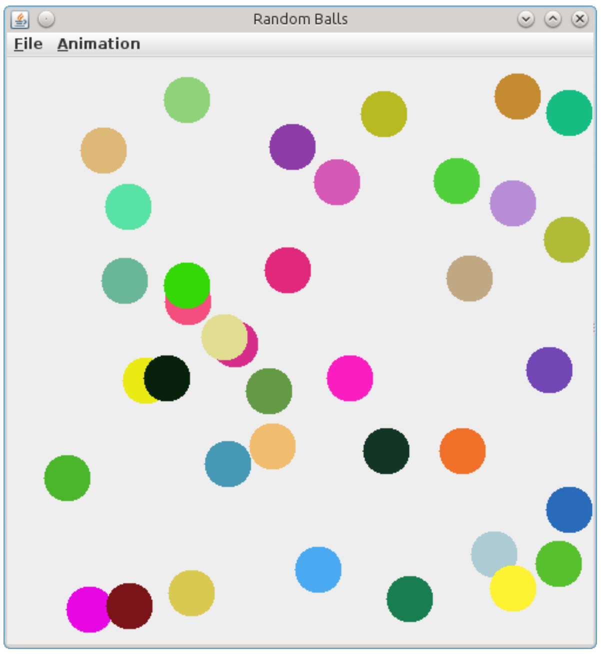
Problem

In this exercise three specific animations need to be implemented.

Random balls

Make a class `RandomBalls` that inherits from the class `AnimationGraphicsFrame`. In each step of the animation, a ball (of a given size) is painted randomly on the graphics panel. Each ball should have a random color. Before and after the animation the graphics panel should be empty.

After some animation steps, the result could be



Be sure that all the generated balls fit completely inside the panel. A general color can be created as follows:

```
Color color = new Color(red, green, blue);
```

where `red`, `green` and `blue` have to be integer values between 0 and 255. The class `Color` is part of the package `java.awt`. This constructor creates a color in the RGB color model. This is an additive color model, meaning that an RGB color is formed by superimposing three light beams (red, green, and blue). Some common colors have the following codes:

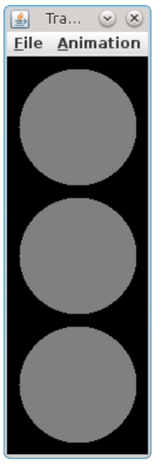
black	(0, 0, 0)
blue	(0, 0, 255)
cyan	(0, 255, 255)
gray	(128, 128, 128)
green	(0, 255, 0)
magenta	(255, 0, 255)
red	(255, 0, 0)
white	(255, 255, 255)
yellow	(255, 255, 0)

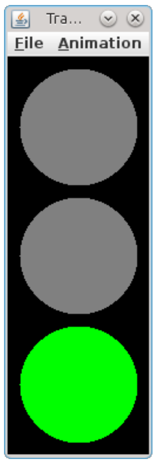
The above colors are also predefined as `Color.BLACK`, `Color.BLUE`, `Color.CYAN`, etc. The method `Math.random()` can be used to generate a random number between 0 and 1.

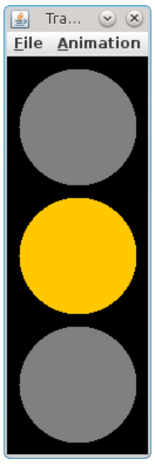
Traffic light

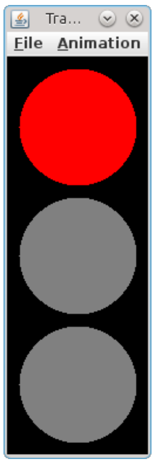
Make a class `TrafficLight` that inherits from the class `AnimationGraphicsFrame`. It should simulate the working of a traffic light: in each step of the animation, the active light should change first from green to orange, then to red, and back to green. Choose the color black as the background color of the traffic light; and a non-active light is colored gray. Before and after the simulation, the three lights are set to non-active (so gray).

The non-active case, and the three active cases are given by:









The dimensions of the graphics window can be manipulated through the method `setGraphicsDimension(width,height)` of the class `GraphicsFrame`. The method `setResizable(boolean)` could be useful as well. The background color can be modified using the method `setBackground(color)` of the class `GraphicsPanel`.

Moving spiral

Make a class `MovingSpiral` that inherits from the class `AnimationGraphicsFrame`. In this animation we are generating an Archimedean (linear) spiral built of small balls. The animation consists of two parts:

1. in the first part, the spiral is extended by painting an extra ball in each step;
2. this is repeated till the spiral reaches the border of the graphics panel;
3. in the second part, the spiral is shortened by removing one ball in each step;
4. this is repeated till all balls are removed, and then go back to step 1.

Note that the Archimedean spiral can be described in polar coordinates (r, θ) by the equation

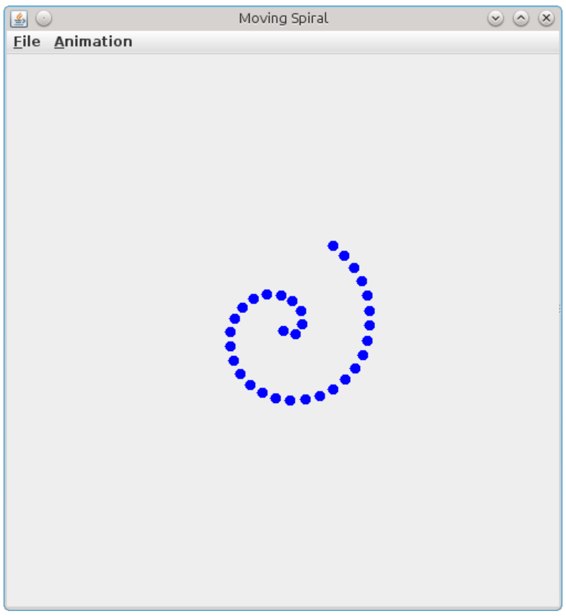
$$r = a\,\theta$$

where a controls the distance between successive turnings. Since we want to center the spiral around the midpoint (x_m, y_m) of the graphics panel, points on the spiral could have the following x/y coordinates:

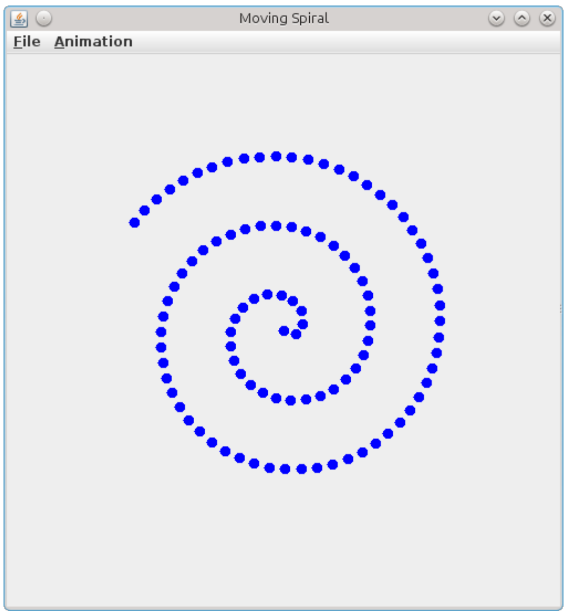
$$x = x_m + a\,\theta \sin\theta, \qquad y = y_m + a\,\theta \cos\theta$$

parameterized by the variable θ.

After some animation steps, in the first part, the spiral of balls increases as follows:



…

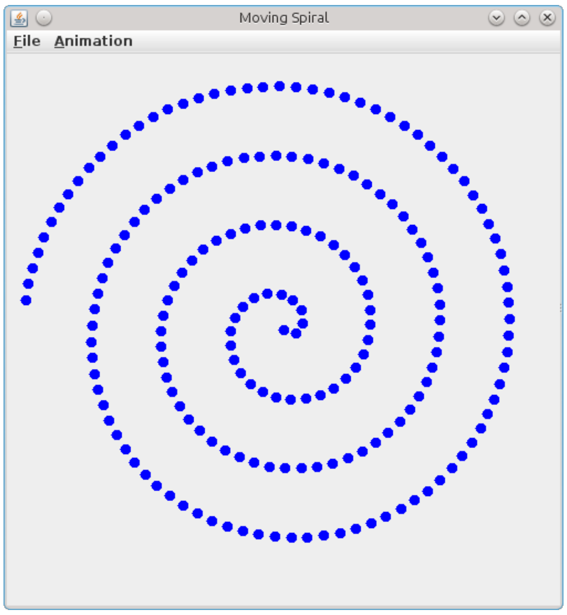


A quite simple formula for obtaining a visually pleasing step size of the parameter θ could be

$$\Delta\theta_k = \pi R / (a (\theta_k + 1.25))$$

where R is the radius of the small balls, and θ_k is the current parameter value.

This spiral growing continues till we reach the border, like



Then, in the second part of the animation, the spiral of balls decreases again (so it follows the opposite movement compared with the first part).

Before the simulation the graphics panel should be empty. After the simulation the complete spiral is painted.