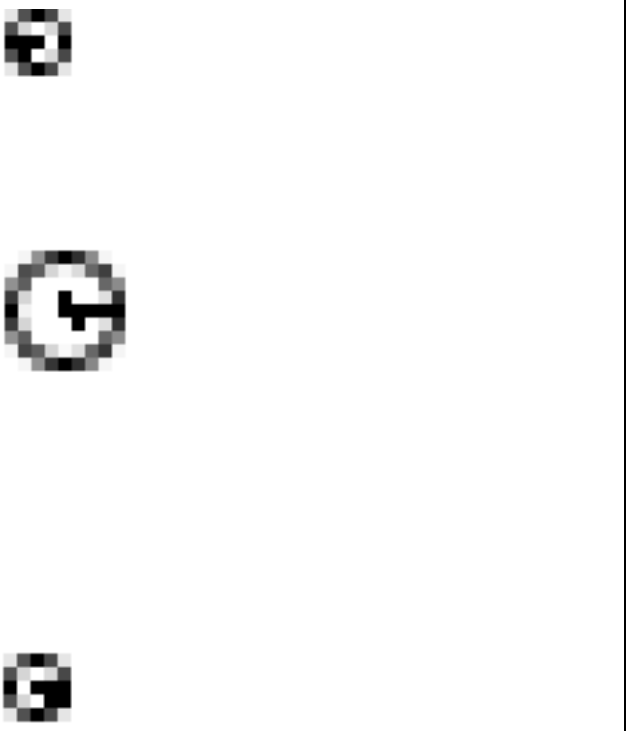
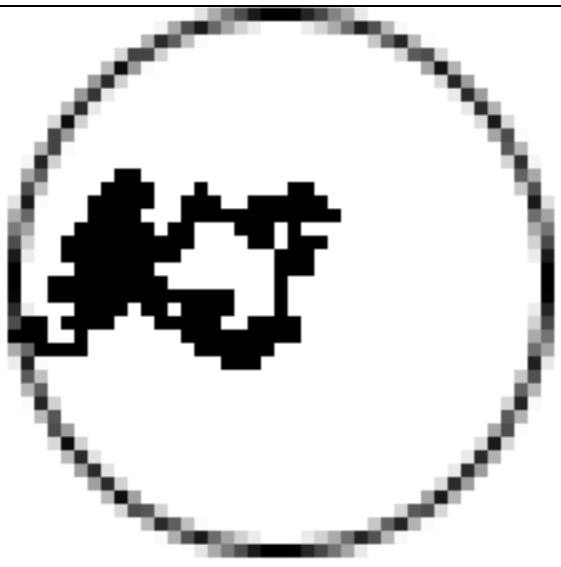
CSE 142, Autumn 2011
Programming Assignment #5: Random Walk (20 points)
Due: Tuesday, November 1, 2011, 11:30 PM

This assignment focuses on using `while` loops to draw two-dimensional random pixel "walks" on a `DrawingPanel`. Two runs of the program are reproduced below with console output on the left and `DrawingPanel` output on the right:

```
Welcome to the CSE 142 random walk simulator.

Radius? 2
Time between steps (ms)? 1000
Walker starting at x=2, y=2
x=1, y=2, steps=1
x=1, y=3, steps=2
x=1, y=2, steps=3
x=0, y=2, steps=4
Walker escaped in 4 step(s).
Walk again (yes/no)? y

Radius? 4
Time between steps (ms)? 100
Walker starting at x=4, y=4
x=4, y=3, steps=1
x=4, y=4, steps=2
x=5, y=4, steps=3
x=5, y=5, steps=4
x=5, y=4, steps=5
x=6, y=4, steps=6
x=7, y=4, steps=7
x=6, y=4, steps=8
x=7, y=4, steps=9
x=8, y=4, steps=10
Walker escaped in 10 step(s).
Walk again (yes/no)? YAP

Radius? 2
Time between steps (ms)? 0
Walker starting at x=2, y=2
x=1, y=2, steps=1
x=1, y=3, steps=2
x=0, y=3, steps=3
Walker escaped in 3 step(s).
Walk again (yes/no)? Nope

Total walks = 3
Total steps = 17
Best walk   = 3
```



```
Welcome to the CSE 142 random walk simulator.

Radius? 20
Time between steps (ms)? 0
Walker starting at x=20, y=20
x=20, y=21, steps=1
x=20, y=22, steps=2
x=20, y=21, steps=3
...
<412 lines of output omitted to save paper>
...
x=0, y=24, steps=416
Walker escaped in 416 step(s).
Walk again (yes/no)? cheese

Total walks = 1
Total steps = 416
Best walk   = 416
```



Program Description:

This assignment focuses on while loops, random numbers, and using objects. Turn in a file named `RandomWalk.java`. Your program will draw a pixel-sized "random walker" that moves in random directions on a `DrawingPanel` until it has moved a certain distance away from its starting location. Read more about interesting properties of random walks here:

- http://en.wikipedia.org/wiki/Random_walk

When the program runs, a 100x100 `DrawingPanel` appears with a white background and an introduction is displayed on the console. The `DrawingPanel` should be zoomed in by a factor of 5 (syntax for doing this is shown later in this document). Then your program should perform one or more random walks. A walk begins by asking the user for the **radius** (in pixels) of the walk area and the time (in milliseconds) between steps. The `DrawingPanel` draws a black-outlined circle with the radius *r* provided by the user, centered at point (*r*, *r*). (In other words, the top-left corner of the circle's bounding box is at (0, 0), and its width and height are twice as large as the radius *r* the user types.)

Next, the steps of the random walk are drawn on the `DrawingPanel` and displayed on the console. On the `DrawingPanel`, the walker is a single black pixel that begins in the center of the circle. At each step, the walker randomly steps its position up, down, left, or right by 1 pixel. The walker should choose between these four choices randomly with equal probability. After each move, a message should print to the console indicating the walker's (x, y) position and the number of steps made so far. Between steps, the walker will pause for the amount of time specified by the user. The walk ends when the walker reaches the perimeter of the circle (i.e., when its distance from the center is greater than or equal to the circle's radius). When the walk ends, the program reports how many steps were made.

The previous output demonstrates your program's behavior. Yours will generate different random steps, but your output structure should match exactly. If you like, you may assume that no run will require ≥ 999,999,999 steps.

After each run, the program asks the user if they want to do another walk. Assume the user will give a one-word answer. The program should walk again if the user's response begins with Y. That is, answers such as "y", "Y", "YES", "yes", "Yes", or "yeehaw" all indicate that the user wants another walk. If the user wants to do another walk, the `DrawingPanel`'s contents clear out to white, and the steps described above repeat. (To clear out the contents of the `DrawingPanel`, use the `DrawingPanel`'s `clear` method.)

If the response does not start with Y, assume that the user does not want to walk again. For example, responses such as "n", "N", "no", "okay", "0", and "cheese" mean that the user doesn't want any more walks. If the user chooses not to walk again, the program prints the total runs, total steps for all runs, and the steps for the best run (the one that required the fewest steps).

These random walk images also look a little like Rorshach ink blot tests. Once you're done, you may wish to optionally save your output as an image, and submit it along with what you think your "ink blot" looks like.

Implementation Guidelines:

Your program must **zoom** everything displayed on the `DrawingPanel` by a factor of 5 to help you see each individual step made by the random walker. Use the `zoom` method from `DrawingPanel`. For example, given a `DrawingPanel` called `panel`, the following line of code will zoom in by a factor of 5:

```
panel.zoom(5);  // zoom the DrawingPanel by a factor of 5
```

This will make each individual pixel appear on the screen as a group of 5x5 pixels. The `DrawingPanel` will do this scaling for you, so you do not need to manually scale any of your drawing commands by 5. The user may also change the zoom level manually from the `DrawingPanel`'s zoom menu. Note that circles do not look very round when zoomed in.

The repetition in this program should be done using **`while` loops**. We suggest that you develop this program in stages:

- Work on a **text-only version** of the program before adding graphics.
- Initially write a version that does **just one random walk**.
- Initially test using **very small radius values** such as 3 or 5.
- Initially test using large values for the time between steps such as 1000 or 2000 (1 or 2 seconds).

Use a single **class constant** for the `DrawingPanel`'s size (default 100). By changing this constant and recompiling, it should be possible to run your program with a larger or smaller window and have all behavior update correctly.

Examine **distances between points** to determine whether the walker has exited the circle. The formula to compute the distance between two points is to take the square root of the sum of the squares of the differences in x and y between the two points. For example, the distance between the points (11, 4) and (5, 7) is $\sqrt{(11-5)^2 + (4-7)^2}$ or roughly 6.71.

One optional strategy you might wish to try that can make the program easier is to represent the random walker's initial position and current position as **`Point` objects** and use the `Point`'s `distance` method so you do not have to write your own. Remember to `import java.awt.*;`

Produce randomness using a single **`Random` object**. `Random` objects produce random integers. These can be mapped to arbitrary random choices. For example, to randomly choose a color between red, yellow, and blue, pick a random integer from 0 through 2, and consider 0 to be red, 1 to be yellow, and 2 to be blue. Remember to `import java.util.*;`

Draw a single pixel by filling a 1x1 rectangle. For example, to draw a pixel a `Point` variable `p`'s coordinates, you'd say:

```
g.fillRect(p.x, p.y, 1, 1);   // draw one pixel at p's position
```

The "**time between steps**" in this program is a short delay that your program should wait between each time it draws the random pixel walker, producing an animation effect. Implement this delay by calling the the `DrawingPanel`'s `sleep` method with a parameter of the number of ms to sleep. For example, the following will pause for 1000 ms (1 second):

```
panel.sleep(1000);   // pause for 1000 ms
```

Assume valid user input. When prompted for numbers, assume that the user will type valid integers in proper ranges (≥ 0 and small enough to fit in the panel). When the user is prompted to play again, the user will type a one-word answer. Read yes/no answers using the `Scanner`'s `next` method (not `nextLine`, which can cause strange bugs when mixed with `nextInt`). If you get an `InputMismatchException`, you are trying to read the wrong type of value from a `Scanner`. To deal with the yes/no user response, you may want to use `String` methods described in Chapters 3-4 of the book.

Produce **repetition** using `while` or `do`/`while` loops. An *inappropriate* way to produce repetition would be to write a method that calls itself, or a pair of methods A and B where A calls B and B calls A, creating a cycle of calls.

The **book** has several helpful sections: Chapter 5's case study is a relevant overall example. Fencepost and sentinel loops from Chapter 5 are also helpful. For the "best game" part of the program, read textbook section 4.2 on min/max loops.

Stylistic Guidelines:

Structure your solution using static methods that accept parameters and return values where appropriate. For full credit, you must have at least **3 non-trivial methods** other than `main` in your program. Two of these *must* be the following:

1. a method to perform a **single "random walk"** on a `DrawingPanel` *(not multiple walks)*
   (a "walk" means taking multiple steps until the walker has gone from the center of the circle to its edge)
2. a method to **report the overall statistics** to the user

Define other methods if they are useful for structure or to eliminate redundancy. Unlike in past programs, it is okay to have some `println` statements in `main`, as long as your program has good structure and `main` is still a concise summary of the program. For example, you can place the loop that performs multiple walks and the prompt to walk again in `main`. As a reference, our solution has 4 methods other than `main` and occupies between 110-120 lines total.

For this assignment you are limited to the language features in Chapters 1-5 of the textbook. Use whitespace and indentation properly. Limit lines to 100 characters. Give meaningful names to methods and variables, and follow Java's naming standards. Localize variables whenever possible. Include a comment at the beginning of your program with basic description information and a descriptive comment heading at the start of each method. Since this program has longer methods than past programs, also put brief comments *inside* the methods explaining relevant sections of your code.

(Optional) Milestone For Free Late Day:

To provide incentive to work on this programming project early, we will offer **one extra late day** to students who submit a partial version of this program with only console output by **11:30pm on Sunday October 30**. The late day can be used on any program with a future deadline, including this assignment. This initial version of the program only needs to produce the output shown on the left side of the sample output grid on page 1 for a *single walk* and does *not* need to include a `DrawingPanel`. Output format must match exactly though your steps will vary based on random values produced. See the course web site for an example log of this milestone version.