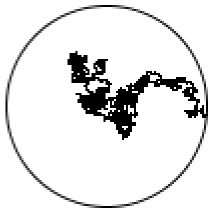
CSE 142, Winter 2008
Programming Assignment #5: Random Walk (20 points)
Due: Tuesday, February 12, 2008, 4:00 PM

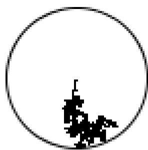
Program Description:

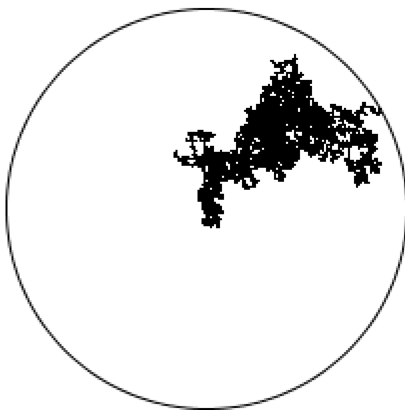
This assignment focuses on while loops, random numbers, and using objects. Turn in a file named `RandomWalk.java`. The program draws a pixel-sized "random walk" that moves in random directions on a `DrawingPanel` until it has moved a certain distance away from its starting location. A random walk visualizes the idea of taking repeated steps in random directions. Read more about interesting mathematical properties of random walks here:

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

```
<< your introduction here >>

Radius? 50
I escaped in 2468 move(s).
Walk again (yes/no)? y

Radius? 35
I escaped in 987 move(s).
Walk again (yes/no)? YES

Radius? 100
I escaped in 13713 move(s).
Walk again (yes/no)? n

Total walks = 3
Total steps = 17168
Best walk   = 987
```







The program begins with an introduction message of your choice. Print anything you like, but keep it to a reasonable number of lines, no offensive remarks, reading from a `Scanner`, infinite loops, etc.

When the program runs, a 500x500 `DrawingPanel` appears with a white background. Then your program should perform 1 or more random walks. A walk begins by asking the user for the **radius** (in pixels) of the walk area. After the user inputs a radius *R*, the `DrawingPanel` draws a black-outlined circle with this radius, whose center is at the 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 `DrawingPanel` draws the steps of the random walk. The walker is a single black pixel that begins in the center of the circle. At each step, the walker randomly moves its position up, down, left, or right by 1 pixel. The walker should choose between these four choices randomly with equal probability. The walk ends when the walker reaches the perimeter of the circle (when it walks so far that its distance from the center is greater than or equal to the circle's radius). When the walk ends, the program reports how many moves were made.

After each game, the program asks the user 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`, fill a white rectangle whose size is as large as the entire panel.)

Otherwise assume that the user does not want to walk again. For example, responses such as "n", "N", "no", "okay", "0", and "hello" would mean that the user doesn't want any more walks. If the user chooses not to walk again, the program prints overall statistics. The total runs, total moves for all runs, and the best run (the one that required the fewest moves) are displayed.

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

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 post this image to Facebook along with what you think your "ink blot" looks like.

Implementation Guidelines:

Your program must use a **class constant** for a `boolean` flag named `DEBUG`. This flag will help you develop your program incrementally and verify its correctness. When `DEBUG` is set to `true`, after each time the random walker moves, a message should be printed to the console indicating the walker's (x, y) position and the number of moves made so far. A series of such messages should match the format below at right. Setting your flag to `false` and recompiling should stop messages such as `x=3, y=2, moves=1` from printing. Turn in your program with this flag set to `false`.

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

- Work on a text 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 always print the "debugging" messages such as the ones at right.

Example output with `DEBUG` = `true`

```
Radius? 3
x=3, y=2, moves=1
x=2, y=2, moves=2
x=2, y=1, moves=3
x=1, y=1, moves=4
x=1, y=2, moves=5
x=1, y=1, moves=6
x=0, y=1, moves=7
I escaped in 7 move(s).
```

You should also use a single **class constant** for the `DrawingPanel`'s size (default 500). 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. However, if you write this program the way we intend, you shouldn't have to manually program this formula yourself. You should represent the random walker's initial position and current position as `Point` objects and use those objects' methods for any relevant computational tasks. 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
```

If you like, you can optionally cause your `DrawingPanel` to animate by calling its `sleep` method with a parameter of the number of ms to sleep. Doing so between drawing operations causes animation as the walker is walking. For example:

```
panel.sleep(5);    // animation delay of 5ms (optional)
```

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. To deal with the yes/no user response, you may want to use `String` methods described in Chapters 3-4 of the book.

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:

- a method to perform a **single "random walk"** on a `DrawingPanel` (not multiple walks)
- a method to **report the overall statistics** to the user

You may 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 80-120 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 comment 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.