Test Program 1

Handout 6
CSCI 134: Fall, 2004
6 October

Due: Friday, 15 October, 2004 at 4PM

Complete each of the following three problems. The handouts web page at

`http://www.cs.williams.edu/~cs134/handouts.html`

contains a link to download a folder containing starter folders and demonstration programs for all three problems.

You are encouraged to reuse the code from your labs and our class examples. Submit your code in the usual way by dragging it into the Dropoff folder for your section. Please do **not** submit three separate folders. Instead, place the folders for all three of your complete programs into one folder and name your folder with your last name followed by TP1. So if your last name is Smith, your folder would be named SmithTP1. Additionally, be sure that your name appears in the title of each of the subfolders, and then place the main folder in the dropoff folder for your lab. Good luck!

Collaboration Guidelines A test program is a laboratory that you complete on your own, without the help of others. It is a form of take-home exam. You may consult your text, your notes, your lab work, our on-line examples, and the web pages associated with the course web page, but use of any other source (human or otherwise) for code is forbidden. You may not discuss these problems with anyone aside from the course instructors, but you should not hesitate to raise any question with them. You may only ask the TA's for help with hardware problems or difficulties in retrieving your program from a disk or network. The use of any other outside help or sources is a violation of the Honor Code.

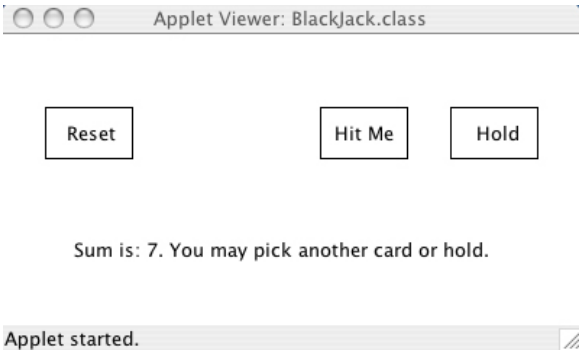
Problem 1: BlackJack

In this question, you will write a program to play a simple version of the card game of BlackJack. In BlackJack, there is a dealer and a player. The game begins with the dealer giving the player two cards. The player sums up the values of the two cards (where face cards count as 10, aces count as 1 or 11, and all numbered cards count as their numbers). Based on the sum of the first two cards, the player decides whether or not to ask for another card (or *hit*). The dealer keeps dealing cards to the player until the player wants to stop (or *hold*). The goal of the game is to have the sum of the value of the player's cards be as close to 21 as possible without going over. If the sum goes over 21, the player loses. Once the player holds, the dealer deals out their own hand and the winner is determined according to the rules below.

- If the player's cards total 21, the player wins.
- If the player's score goes over 21, the player loses.
- If the player's score is less than 21, the score is compared to the dealer's score:
  - If the player's score is greater than the dealer's score, *or* the dealer's score is over 21, then the player wins.
  - If the player's score is *equal* to the dealer's score, it is a tie, which is often called a *push* in gambling terminology.
  - Otherwise, the player loses.

Your game will simplify some of these rules and is described below.

The program will be a single class which extends WindowController (with a size of 400 by 200) that looks something like this:



Your board should (minimally) have these components:

- **Hit Me Button** When the player clicks on the Hit Me button, they receive another card and their score is updated accordingly. If they go over 21, they lose.
- **Hold Button** When the player clicks the Hold button, your program must decide whether the player wins or loses, based on the score of the dealer's hand vs. the score of the player's hand.
- **Reset Button** When the player clicks on the Reset button, two cards are generated and the player's score is reset accordingly.

The game proceeds as follows:

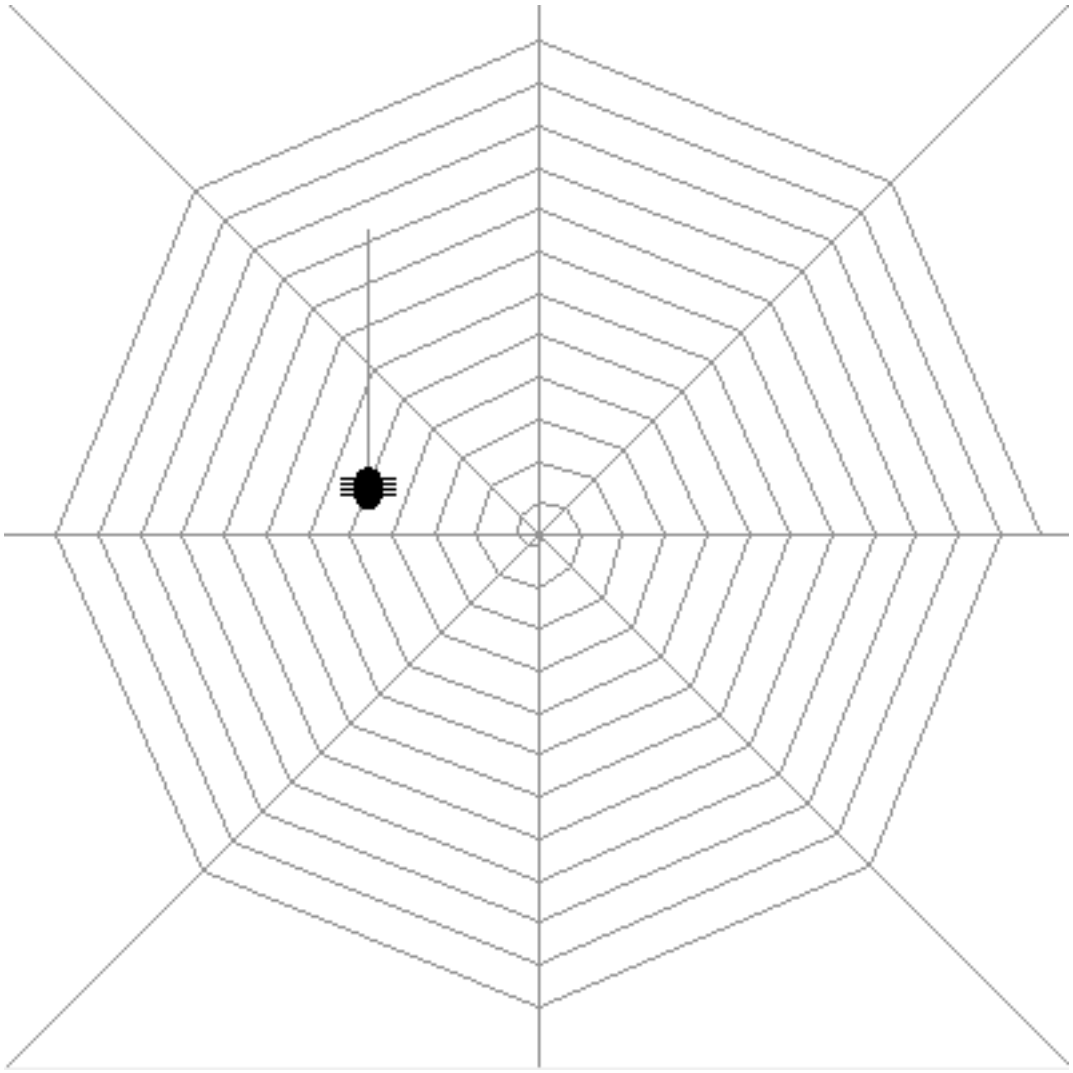
1. Play begins with the player receiving two cards from the dealer and the sum of those being displayed. Cards in our game will just be integers in the range 1 to 10, and you may use a `RandomIntGenerator` to "deal" new cards as needed. In other words, we will not worry about face cards, aces, etc.
2. The player then has two options, they can ask for another card by clicking on the "Hit Me" button which adds another random card value to the sum, or they can stop by clicking the "Hold" button.
3. If the player requests another card, its value is added to the score. If the hand is still under 21, the player may continue to ask for more cards. If the hand sums to 21, the player wins and an appropriate message is displayed. Otherwise, the player loses and a message indicating the loss should be displayed.
4. If the player holds, the dealer's score is computed by generating three random card values and adding them. The player's score is then compared to the dealer's, and the winner is determined according to the scoring rules above.
5. The player clicks on the "Reset" button to begin again after winning or losing. (You may notice that the player can still press the "Hit Me" and "Hold" buttons in our demo version, even after the winner has been determined. It is fine if you program behaves in this way, or you could think about how to disable the buttons when they should not be used.)

Extra Credit There are many embellishments you may make to get extra credit. A sample of some options is given below. When your program generates a card, it generates a random integer between 1 and 10, essentially ruling out face cards and treating aces as 1. For extra credit, try making a better "deck" that generates random numbers between 1 and 13, where 11, 12, and 13 represent face cards that count as 10 toward a hand's score. Similarly, you could change your program so that any ace cards (ie, cards with value 1) can be treated as having the value of 1 or 11.

In our Blackjack, the dealer always takes three cards. In real Blackjack, the dealer takes two cards, and then continues to take additional cards until the dealer hand sums to 17 or higher. You could also implement these rules for generating the dealer's hand.

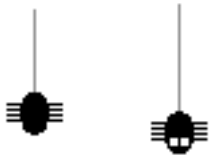
Problem 2: Hanging By a Thread

The question asks you to write a program that animates a simple spider.



The canvas should initially be empty, except for the spider web background. (The starter code reuses our spiral program to draw the web for you.) When you press the mouse in the window, the spider appears centered at the location where you clicked. As you drag the mouse, the spider should descend on a gray thread that stretches as the spider moves down the screen. After descending for a while, the spider turns around and climbs back up the thread to where it began. On the climb up, the spider's eyes should be open and its thread should disappear as it climbs. When the spider again reaches where it started, it turns around and start its track down and up again.

Here are close-ups of what our spider looks like with its eyes open and closed. Feel free to enhance the spider's appearance in any way you like (provided that it behaves according the rules we have described).



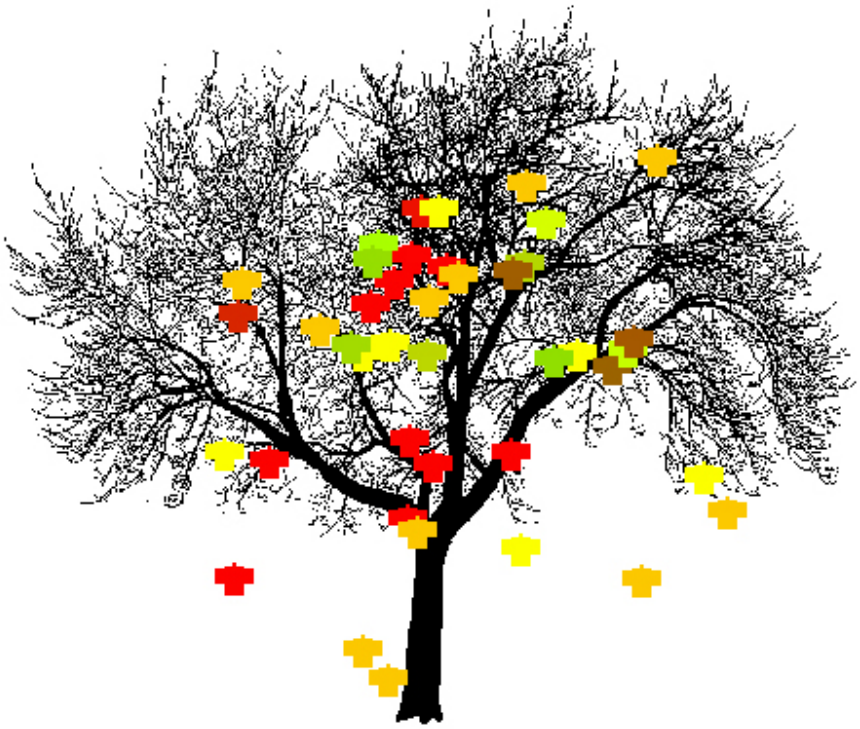
When the mouse is released, the spider and its thread disappears. Pressing the mouse at a different location will cause the spider to appear at that position.

You should set the run configuration to size 400 by 400. Your program should be divided into two classes: a window controller called `SpiderController` and a `Spider` class. *You will only need to modify the* `Spider` *class.* We have written the `SpiderController` class for you. For the `Spider` class, he `Spider` constructor should expect three parameters: the Location where the spider should appear, a double specifying the how far the spider will descend, and the canvas. The `Spider` class should define the following methods used by the controller to implement the functionality described above:

- `public void move(double distance)` moves the spider by the specified distance. The spider should keep track of whether it should move up or down by that amount.
- `public void removeFromCanvas()` makes the spider disappear from the canvas.

The spider will only move when the mouse moves, so `Spider` should not extend `ActiveObject`.

Problem 3: Falling Out of a Tree



This program is a rendering of the changing seasons in Williamstown. The program begins with a tree containing no leaves. (We will provide a suitable picture, but feel free to construct or find

your own.) When you drag the mouse over the tree, green leaves appear at the mouse's location. These leaves should gradually change to fall colors— red, orange, or yellow for example— and then the leaves should fall off the tree and float to the ground, where they gradually disappear. Our leaves disappear gradually by shrinking the height of their rectangles until they become very short.

Continuing to drag the mouse around on the branches of the tree creates additional leaves. As illustrated in our sample, you should be able to place leaves over most of the tree, but not in the sky. You will need to decide how to guarantee that leaves are only put in reasonable places.

Your program should be divided into two classes: the `Fall` window controller class and the `Leaf` active object class. The `Fall` window controller handles the mouse interactions, and the `Leaf` class describes how a single leaf should behave. We will leave most of the design up to you, but be sure to pass to the `Leaf` constructor sufficient information to describe where the leaf should be placed, what color it should become, and how far and fast it should fall. Feel free to embellish any aspect of this program.

A few details to keep in mind:

- Implement this program in small pieces. A reasonable first step is to have a single green leaf appear on the tree when the mouse is clicked. Then add the falling behavior, and finally add the color transitions and disappearing. One you have one leaf working, then go back and change the window controller to create leaves as you drag. Even if you get stuck on one part, try to implement as many of these pieces as you can.

- You should design your program so that different leaves change color and fall at different speeds.

- There are many ways to transition from one color to another over time. A simple way to transition from, say, color1 to color2 is to combine the red, blue, and green values of each color with different weights. If `color1 = new Color(`$red1, green1, blue1$`)` and `color2 = new Color(`$red2, green2, blue2$`)`, then the color with the following RGB (Red-Green-Blue) values is the same as `color1`:

$$\begin{aligned} red &= 1.0 * red1 + 0.0 * red2 \\ green &= 1.0 * green1 + 0.0 * green2 \\ blue &= 1.0 * blue1 + 0.0 * blue2 \end{aligned}$$

  Similarly, the color with the following RGB values is "half-way" between `color1` and `color2`:

$$\begin{aligned} red &= 0.5 * red1 + 0.5 * red2 \\ green &= 0.5 * green1 + 0.5 * green2 \\ blue &= 0.5 * blue1 + 0.5 * blue2 \end{aligned}$$

  You may wish to use the `getRed()`, `getGreen()`, and `getBlue()` methods for `Color` objects in this part. These three methods return a color's integer values for the three primary colors.

- Depending on how you implement the color changing, you may find that you wish to assign a `double` value to an `int` variable or pass a `double` value as a parameter where an `int` is expected. If you try to use a `double` in these ways, Eclipse will complain. To convert a `double` value to an `int` value, you must use a type cast, as illustrated by the following line:

  ```
  int x = (int)(0.55 * y);
  ```

  This line multiples y by 0.55 to obtain a real number and then truncates the number to convert it to an integer. See page 150 of the text book for more details. Also, keep in mind that dividing an integer by integer yields an integer result. Thus, 2/5 = 0, whereas 2.0/5=0.4.

Grading Guidelines

The test programs will be scored out of 100 points, divided evenly among the three programs. For each program, half the points will be awarded for style, and half the points will be awarded for correctness. Some specific style items we will look for include:

Proper use of boolean conditions
Proper use of ifs/whiles
Appropriate variable declarations (instance vs. local, public vs. private)
Descriptive comments
Good names
Good use of constants
Appropriate formatting (indenting, white space, etc.)
Parameters used appropriately

The following is a brief list of the key correctness issues, although you should be sure to try to implement all features list above.

BlackJack	Display all the buttons correctly with text labels Hit me button adds to sum Hold button stops game Scoring works correctly Reset works correctly
Hanging By a Thread	Initial spider and thread correctly Spider climbs down properly Eyes open and spider climbs properly Spider erases itself
Falling Out of a Tree	Tree placed correctly Dragging creates green leaves on tree Leaves fall properly Leaves vary in color and speed Leaves change color properly

As with in the lab assignments, even programs that are not entirely correct or that do not contain all of the required features can receive a large fraction of the points, provided that they are well written and documented.

You may earn up to four extra points of extra credit as well for adding interesting features to your programs.