

## CS 51G Laboratory # 10

### Apples

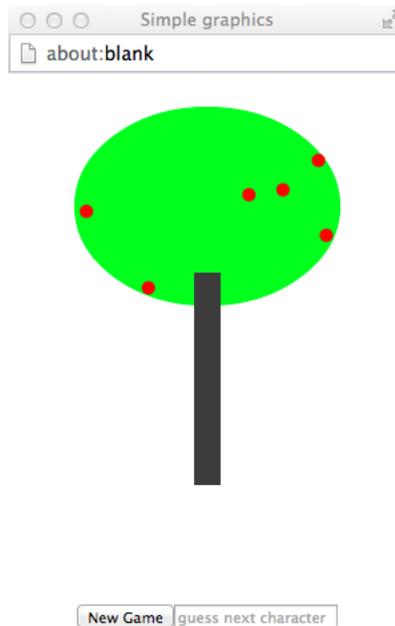
**Objective:** To practice using String methods.

In this lab we'll be implementing a version of the game Hangman. **You are encouraged to work in pairs in this lab.** If you decide to work with a partner, the two of you must do *all* of the design and coding together. A pair should turn in only one copy of the program, but make sure both people's names are on the folder and in the comments.

Hangman is a simple game in which one person, the "*supplier*", thinks of a word and another person, the "*guesser*", tries to guess it. The game starts with the *supplier* writing down a row of dashes, with each dash representing one letter in the word. The *guesser* then repeatedly guesses a letter. If the letter appears in the word, the *supplier* replaces each appropriate dash with that letter. If the letter doesn't appear in the word, the *supplier* makes a note of that. The *guesser* is allowed some number of incorrect guesses before they lose the game. In our case they'll be allowed 6 incorrect guesses.

The name Hangman comes from the fact that instead of simply noting that the *guesser* has lost because they've guessed 6 letters that don't appear in the word, traditionally the *supplier* has drawn a feature of a hanging person for each incorrect letter, with the *guesser* losing when the complete figure of a hanging person has been drawn.

In our case we'll be playing a (G-rated) version of the above game called Apples. In this version, the *supplier* (your program) draws a tree with 6 apples, and each incorrect guess by the *guesser* causes one apple to fall off the tree. When all the apples are gone, the *guesser* has lost.



Guesses in either uppercase or lowercase are acceptable. Furthermore, nothing happens if you guess a letter that you've already guessed before.

## Overview

Your program will consist of two classes plus two objects:

**dictionary** will be an object that is responsible for returning a random word to be guessed.

**apples** will be the object corresponding to the main program. It sets up the game interface.

**appleGame** will be the class that maintains the state of the game.

**appleGameDisplay** will be the class that maintains the image of the apple tree.

We will provide complete implementations of the **dictionary** object and the **appleGameDisplay** class, which are described in the following section. You'll need to write the other two, following the specifications below.

**The dictionary class** The only thing you need to know about the **Dictionary** class is that it has one method:

```
getRandomWord -> String
```

**The AppleGameDisplay class** This class deals with drawing the tree with the apples and provides methods that let you remove an apple and that let you refill the tree. The type is:

```
type AppleGameDisplay =
  fillTree -> Done
  numApplesLeft -> Number
  removeApple -> Done
```

The class header is

```
class appleGameDisplay.apples(numApples: Number) on (canvas: DrawingCanvas)
  -> AppleGameDisplay
```

You can read the comments in the provided code to see more what these methods do, but most should be intuitive from their names.

Both the **dictionary** and **appleGameDisplay** are in file **AppleGameDisplay.grace**.

**The appleGame class** The **appleGame** class will manage the state of the game. This involves three basic things:

- You need to keep track of the state of the game. You may **NOT** use arrays to keep track of things. Instead, you should use **Strings** and appropriate string operations to keep track of the state. Don't forget that **Strings** are immutable!

There are three basic things you'll need to keep track of: the word the person is trying to guess, the current state of the dashed word (i.e. the *guesser's* current partial knowledge of the word) and what letters have been guessed so far.

- Generate and update the state of the tree using an **AppleGameDisplay** object.
- Generate and update the textual display of the game.

The class for `appleGame` should have the following header:

```
class appleGame.apples(numApples: Number)
    on (canvas:DrawingCanvas) -> AppleGame
```

where

```
type AppleGame =
    newGame -> Done
    guessLetter(letter:String) -> Done
```

Method `guessLetter` takes a character `letter` and, if it hasn't already been tried, determines whether or not it appears in the word, and updates both the text and the picture accordingly (remember the guesses should be case insensitive).

Method `newGame` resets both the text and tree display, and chooses a new word from the dictionary. That is, you should be able to play as many games as you like without restarting the program.

**The apples object** This object manages the overall game. The constructor should set up the GUI so that there's a `Button` labeled "New Game" or and a `TextField` for entering guesses.

The user interacts with the game by entering a letter into the field and pressing enter. If the string entered is more than one character or is not a letter then the program should not respond, and should again prompt the user to enter a letter.

## Design

As usual we will require that you prepare a written "design" for your program before lab. You only need to provide a design for the `appleGame` class and `apples` object since we provide the code for the other object and class. At the beginning of the lab, we'll come around and briefly examine each of your designs to make sure you are on the right track. At the same time, we'll assign a grade to your design.

## Implementation

As usual, we suggest a staged approach to the implementation of this program. This allows you to identify and deal with logical errors quickly. The size of your applet should be 500 pixels wide by 500 pixels tall.

- Start by getting the initial GUI setup so that the textfield and the button are in the right place.
- Now write the initialization code and the `newGame` method for the `appleGame` class, ignoring the tree display. Make sure you can choose a new random word from the dictionary, and that you can display the right number of dashes. Add in the action to be taken in the `apples` object when the button is pressed and verify that the `New Game` button causes a new word to be chosen.
- Next write the `guessLetter` method for the `appleGame` class. Again, ignore the tree display, and just make sure that your code correctly determines whether the letter appears in the word and that it updates the text display appropriately. Add in the action for the textfield and verify that the GUI works correctly.
- Finally, add in the code that uses the `AppleGameDisplay` class.

The challenging string manipulation question for this lab is how to update the dashed word when a new letter is entered. There are many ways to do this, one reasonable way is to first check if the letter appears in the word at all. If so, iterate through the characters in the word to be guessed, and if the character occurs in it, then update the dashed word. To do this, you can build up a new, temporary, dashed word as you go along, then when you get to the end replace the original dashed word with the temporary one. Think about this problem before beginning coding the `guessLetter` method! Also don't forget that a letter may appear in a word multiple times.

## Submitting Your Work

The lab is due at 11 p.m. on Monday as usual. When your work is complete you should deposit in the dropoff folder a folder that contains your program and all of the associated files. Make sure the folder has your name and lab number on it (if you worked with a partner, include both names). Also make sure that your name(s) are included in the comment at the top of each Grace source file that you wrote.

Before turning in your work, be sure to double check both its logical organization and your style of presentation. Make your code as clear as possible and include appropriate comments describing major sections of code and declarations.

Table 1: Grading Guidelines

Value	Feature
<b>Design preparation (4 points total)</b>	
2 points	<code>apples</code> object
2 points	<code>appleGame</code> class
<b>Syntax style (5 points total)</b>	
2 points	Descriptive comments
1 points	Good names
1 points	Good use of constants
1 point	Appropriate formatting
<b>Semantic style (4 points total)</b>	
1 point	Conditionals and loops
1 point	General correctness/design/efficiency issues
1 point	Parameters, variables, and scoping
1 point	Miscellaneous
<b>Correctness (7 points total)</b>	
1 point	GUI works
1 point	game stops and restarts correctly
3 points	handles letters input correctly
1 point	text display correct
1 point	tree display correct
<b>Extra Credit (2 points maximum)</b>	
.5 point	Better text/tree graphics
.5 point	Better interface