Silly Premise

Poor Pop is tired of his children hopping on him. While he does have abs of steel, there is only so much an anthropomorphic bear can take! But his kids want to keep hopping so they can one day have calves as magnificent as Andy’s. Help Pop by creating a DoodleJump game with an infinitely scrolling board so his enthusiastic children can hop to their heart’s content!
New Concepts Covered

- For loops
- `java.util.ArrayList`
- Java generics
- More JavaFX!

Collaboration Policy Reminder

From the collaboration policy:

Collaboration on project design is not allowed, except for during the design discussion for this project under the supervision of your discussion leader TAs. Otherwise, no collaboration is allowed on project-specific details. You are not allowed to discuss the classes you will be using in your project, what methods you will be writing, inheritance hierarchies, the design discussion mini-assignment, or any other design components of the program. You may not discuss implementation or debugging of code for projects with anyone except the course staff.

Importantly, though, you may absolutely discuss general (i.e., not assignment-specific) CS15 concepts with anyone, including other current students. The following falls into this category:

- Going over CS15 lecture slides, our (non-assignment) handouts, Javadocs, etc.
- Discussing object-oriented programming concepts, such as polymorphism
- General syntax questions. For example, “How do I declare an instance variable?”
- How to work remotely, and how to move and hand in files

Note that in each case, any examples used must be from the lectures or your own creativity – you may not discuss how even broad design concepts like containment pertain to a specific assignment.

Mini Assignment

Before starting to code your program, you should think through your design carefully by completing the design discussion mini assignment! This will help you think through your containment and inheritance relationships as well as some of the larger algorithms that you will code in this project. A PDF with your mini assignment solutions is due by the deadline at the top of the handout via cs015_handin DoodleJumpDiscussion.
Assignment Specifications

Before you read this section, make sure you play around with the demo! Better yet, have it open while you read this and play along. Doing so will make understanding the assignment much easier.

In this program, you will write the CS15 version of the popular game Doodle Jump. This game features a “doodle” whose goal is to hop along a series of never-ending platforms without falling to the ground. The doodle will fall according to the laws of gravity (more on this later) until it hits a platform, at which point it will bounce up until gravity brings it down again. The doodle’s horizontal position can be controlled with the left and right arrow keys.

As the doodle jumps higher and higher, the game will need to scroll vertically. By scroll vertically, we mean that when the doodle hits a certain height (say the midway point of the window), rather than have the doodle move higher on the screen, all the platforms should move lower (giving the illusion that the doodle is still climbing up). See the demo for an example.

As the game scrolls upward, old platforms need to disappear off the bottom of the screen and new platforms need to be regenerated at the top of the screen so that the doodle will have more places to jump.

The game ends when the doodle falls past the bottom of the screen because it missed all the platforms as it fell. A message should be displayed to tell the user that the game is over, and at this point the doodle can no longer be moved left or right.

Your game should also include a quit button that closes the game window.

Helpful Design Hints

This program is large and complex, so it is more important than ever to design all your classes before you write any code. We mean it!

Well-chosen relationships between your classes will be very important. You have some experience with making a GUI from Cartoon, so remember what you have learned when creating the GUI components. We have provided you with an App class that takes care of the mainline, but it is your job to create a layout using javafx.scene.layout classes as well as any other graphical components you might need. Once you have an idea of how you want your GUI to look, you should think about the communication between the GUI, the Doodle, and the platforms. Furthermore, you have some experience with Timelines in Java from Cartoon, which will come in handy for animation and physics simulation.
Another important design decision will be choosing how to generate your platforms. The game would be pretty boring if the same platforms were always generated, so you’ll want to incorporate some randomness into your platform generation. At the same time, your platforms should be generated in such a way that the doodle will definitely be able to get from one platform to the next, so making platform generation completely random won’t work either. Your platforms should be generated “semi-randomly,” meaning that their location should be chosen randomly given a certain set of constraints.

So how will you keep track of your platforms? By using a `java.util.ArrayList`, that’s how! An `ArrayList` is a data structure that allows you to store and access a dynamic number of objects in a “list” of sorts, which is exactly what we’re looking for: a way to keep track of all the platforms in our game.

But wait! Why can’t we use an array for this? Since the horizontal and vertical positioning of platforms is based on a degree of randomness, varying numbers of platforms may fit on the screen at any one time. The `java.util.ArrayList`’s dynamic capacity lends itself very well to handling this type of dynamic storage, whereas an array of fixed capacity would not. `ArrayList`’s also allow you to easily add and remove elements.

The Javadocs for the `java.util.ArrayList` data structure highlight some of the important methods that have already been defined for `ArrayList`s, so you should become familiar with these before starting this project.

For a review of `ArrayList`s, look over the later slides in the Arrays lecture. This will also help you understand how to declare and instantiate an `ArrayList` that holds a unique type of object. Lab 6, which will happen next week, will also provide extra practice with `ArrayList`s.

**Physics Simulation**

To model the effects of gravity on the doodle, we’ll need to give it some physical properties, such as position and velocity. We will need to modify these properties over time, using a couple of the handy equations of motion you may have learned in high school physics.
These equations may seem intimidating, so let's break them down further:

We need to calculate a new y-velocity (the rate at which the Doodle is rising/falling), which helps us calculate the new y-position for our Doodle at the end of each KeyFrame. This repeated updating simulates the effects of gravity on a falling Doodle; each successive update will need to build on the velocity and y-position that was computed in the previous update.

The duration of your KeyFrame represents $\Delta t$ in the equation above. Using the duration and acceleration (gravity), you can calculate the new velocity and position of the doodle. Here's pseudocode for updating the velocity and position of the Doodle each time the Timeline cycles through to the end of your KeyFrame:

\[
\text{updated velocity} = \text{current velocity} + \text{ACCELERATION} \times \text{DURATION} \\
\text{updated position} = \text{current position} + \text{updated velocity} \times \text{DURATION}
\]

**Note:** We've converted the original equations to discrete (non-continuous) equations because the approximation is close enough for our purposes.

The constants `ACCELERATION` and `DURATION` are already defined for you in the `Constants` class, though you are welcome to change them. Be sure to review the Math/Making Decisions lecture for more information on constants!
ACCELERATION corresponds to the gravity constant. In the real world, it’s \(-9.8 \text{ m/s}^2\). Because our coordinate system defines the positive y-direction as downward, our acceleration constant is positive (approximately 1000 pixels/s\(^2\)). The DURATION constant corresponds to the amount of time that elapses for each iteration of your KeyFrame and should be used to initialize your KeyFrame, as well as for updating position and velocity.

Be careful of the units, though—DURATION is in milliseconds! You’ll want to use units of seconds in your equations for calculating the doodle’s new velocity and position.

**Remember:** When you're converting the DURATION to seconds you should compute

\[((\text{double}) \text{ DURATION})/1000 \text{ or } \text{ DURATION}/1000.0\] instead of DURATION/1000 to avoid integer division truncation.

Once you have this set up, making your doodle jump is more straightforward! You can detect collisions between your doodle and a platform using javafx.scene.Node intersects method, described in pseudocode here:

```java
boolean intersects(double x, double y, double width, double height) {
    Consider a rectangle at (x, y) with given width and height
    If the node intersects the rectangle:
        return true
    Otherwise:
        return false
}
```

To make your doodle jump, set its y-velocity to the REBOUND_VELOCITY constant (provided in the Constants class) whenever it hits a platform. Be careful to make sure that you're only checking for platform collisions when the doodle is falling down.

**Timeline Updates**

By this point it should be clear that your Timeline is going to have to do a bit more than it did in Cartoon. To make your lives easier, we have provided a list of steps that need to be taken at the end of each KeyFrame. Remember that your EventHandler doesn’t need to do all of this directly. It can (and should) delegate appropriate tasks to other methods and objects. Inside your handle(...) method (or its helper methods!), you should:

1. Calculate a new y-velocity for the doodle
2. Calculate the change in your doodle’s height (\(\Delta h\)) based on a new y-position
3. Check to see if the doodle is above the horizontal middle of the window
4. If the doodle is above the middle:
   move all platforms down by \(\Delta h\)
Else:
   move the doodle up by \(\Delta h\)
5. If the doodle intersects with a platform AND doodle is falling:
set the doodle’s y-velocity to \texttt{REBOUND VELOCITY}.

**Keyboard Interaction**

Keyboard input in DoodleJump should be similar to the key handling that you wrote for Cartoon -- the Cartoon handout has more information on specifics of handling key input, but here are some helpful reminders on handling key input in JavaFX:

- Make sure to \texttt{consume} your \texttt{KeyEvent}s after you are finished with them to prevent the \texttt{KeyEvent}s from triggering other actions in your program.

- For the keys to receive input, you need to “request focus” on the \texttt{Pane} that the key handler is added to. “Focus” deals with which graphical element is currently “selected.” To register and handle key input, we want to give focus to the \texttt{Pane} associated with the key handler. This can be accomplished by calling \texttt{Pane}'s \texttt{requestFocus()} method \texttt{after} the \texttt{Pane} is visible (a.k.a. \texttt{after stage.show()} is called). Think about how you can use delegation to request focus on a \texttt{Pane} contained in another class from your \texttt{App} class! Alternatively, you can call \texttt{setFocusTraversable(true)} on your \texttt{Pane} after you add the key handler to your \texttt{Pane} using \texttt{setOnKeyPressed}, and call \texttt{setFocusTraversable(false)} on all other \texttt{Pane}s and buttons.

**Platform Generation**

Begin by creating the bottommost platform so it is just below your doodle’s start position. Next, you’ll want to generate the platform above that one. You should decide on minimum and maximum x and y distances between two consecutive platforms to ensure that the doodle is able to jump from one to the other.

Here’s an example of how to use this information to generate the platform’s location:

\begin{itemize}
  \item Assumption: Platform \textit{p} has x-y coordinates (300, 600)
  \item Assumption: Doodle can move up 100 pixels and left or right by 200 pixels per jump before starting to fall. (This will differ based on the constants you use, make sure your game is playable)
\end{itemize}

With the above assumptions, the maximum x-distance between consecutive platforms should be 200 and the maximum y-distance should be 100. This means the second platform’s x-position must be a random value between 100 and 500 and its y-position should be a random value between 500 and 600.

**Note:** Platforms should not generate outside the board. Consider the pane’s width when constraining your generation to stay within the board.
The third platform will need to be positioned relative to the second platform, and so on. Do you see a pattern?

How should you generate all the platforms? A loop! At each iteration of the loop, you are creating a new platform whose position is relative to the platform created in the previous iteration of the loop.

Here’s an idea of how you might want to implement this:

1. Create your first platform and store it in a current topmost platform variable. Position the platform directly under the Doodle’s starting location (probably somewhere near the bottom of the Pane).
2. Add that platform to your ArrayList of platforms.
3. While the current topmost platform’s y position is not above the top of the Pane:
   a. Create a new platform and add it to the ArrayList.
   b. Randomly position the new platform relative to the current topmost platform.
   c. Update the current topmost platform variable to reference this new platform.

Remember: To graphically (visually) add or remove Nodes, you should add or remove them from the Scene Graph; to add or remove Nodes in your game logic, you should make sure the appropriate references to the Node (or the object that contains the Node) exist or don’t exist (in the case of removal).

Requirements For Full Credit

There’s plenty of room for creativity in this assignment, but before you do any extra credit, make sure your game meets all of the following specifications!

1. It has a Doodle! (duh...) The Doodle should be its own class - this makes it easier for your doodle to store and calculate relevant information. Remember: delegation!
2. You have a Timeline. At the end of your KeyFrame, the doodle’s velocity is calculated. The doodle should move according to motion equations and the Timeline.
3. The doodle can be moved left and right with the left and right arrow keys.
4. You have platforms (instances of a javafx.scene.shape.Shape, or you can write your own Platform class containing a Shape). The position of each new platform is semi-random. It should be constrained within a certain distance of the platform below it so it is reachable.
5. When the doodle is falling and intersects a platform, it bounces.
6. The platforms scroll down when the doodle reaches the approximate midpoint of the screen.
7. As the doodle jumps higher and the game scrolls, new platforms are created so that the doodle can continue to climb indefinitely.
8. If the doodle falls past the bottom of the screen, a game over label appears and the doodle can no longer be moved with the arrow keys.
9. The game should display a quit button that calls `System.exit(0)` or `Platform.exit()` when clicked.

**Bells and Whistles**

Once you’ve met all of the above requirements, feel free to snazz up your DoodleJump with some bells and whistles! (check out the snazzy demo for inspiration). Reminder: you are not eligible for extra credit unless your project meets minimum functionality requirements, defined at the bottom of the handout. Here are a few ideas:

- Different levels of difficulty
- A button to restart the game
- Make the Doodle a composite shape
- Keeping track of the player’s score
- Support for multiple players
- ‘Smooth’ left/right movement with the arrow keys
- Different types of platforms - movable, breakable, high-jump
- Toys to alter the Doodle’s velocity - jetpacks, helicopter-hats, etc.
- Venomous black holes and hungry monsters

Keep in mind that most TA’s have not implemented every one of these extra credit features, so we cannot guarantee support for them (though they will try send you in the right direction). As with Cartoon, we are unable to give extra credit for sound, as the department computers’ do not have the proper support libraries.

**Coding Incrementally**

Figuring out where to begin on this project can be a bit overwhelming. It will make your life a lot easier to get the program working a little bit at a time. Please don’t try to write a ton of code and then run it! You’ll just cause more headaches for yourself later. **TAs will turn you away from hours if you have not demonstrated that you have carefully attempted each part of the assignment before moving on to the next step.** We suggest following these steps:

1. Make your game window show up.
2. Get your doodle to display.
3. Set up the Timeline that will be in charge of updating the doodle’s location and displaying the graphical changes. (To test if your Timeline is working, you can start with printlines. Afterwards, you may want to move the doodle slightly at the end of your KeyFrame.)

4. Set up the key event handler so you can use the left and right arrows to get your doodle to move.

5. Add some physics simulation so that your doodle falls.

6. Start with creating one platform and add and implement/test collision detection with that platform.

7. Generate a whole screen of semi-randomly positioned platforms so that your doodle can jump its way upwards!

8. Add the vertical scrolling so that when the doodle tries to pass a certain height, it stops moving, and all the platforms move downward. (WARNING! This step is tricky! Really think about the best way to implement this -- careful design and pseudocode will simplify this step greatly.)

9. As platforms scroll down, be sure to delete them and generate new ones both graphically and logically.

Minimum Functionality Requirements

MF Policy Summary: In order to pass CS15, you will have to meet minimum functionality requirements for all projects. If you don’t meet them the first time around, you may hand the project in again until you succeed, but you will keep your original grade. MF requirements are not the same as the requirements for full credit on the project. You should attempt the full requirements on every project to keep pace with the course material. An ‘A’ project would meet all of the requirements enumerated in the assignment specification section of the handout and have good design and code style.

To meet minimum functionality for DoodleJump:

- A doodle and platforms appear on screen.
- The doodle’s position and velocity are updated continuously with a Timeline.
- The doodle can collide with and bounce off of platforms.
- Platforms scroll downward when the doodle reaches some point on the screen.
- The doodle moves left and right in response to key presses.