DoodleJump

Help Session
Help Session Topics

- Design
- Incremental Coding
- Physics Simulation
- Key Input
- ArrayLists
- Platform Generation
- Vertical Scrolling
(One option for)

Containment

[Diagram of Containment structure with classes and methods]
Why use a Game class?

- Effective object-oriented programming is focused on *modeling*
  - How can we model properties and functionality as a system of classes?
- Imagine an incredibly complicated game like chess.
  - Would we want the thousands of line of game logic to all live in the PaneOrganizer class?
  - PaneOrganizer has a clear, logical modeling job - it should handle the overall organization of Panes and the relevant graphics
  - Important application of *abstraction*.
Inheritance
1. Get a window with your Doodle to show up.
2. Add `Timeline` to update your Doodle’s location (you just did this in Cartoon—you know what’s up!).
3. Set up your `KeyHandler` so that the arrow keys move the Doodle left and right.
4. Add physics simulation so that your Doodle’s velocity and position are updated every `DURATION`.
5. Create one platform and get collision detection working.
6. Generate more platforms semi-randomly.
7. Add vertical scrolling!
8. Add `Label` to tell user the game is over.
Your Doodle shouldn't just move up and down at a constant speed, it should *accelerate* or *decelerate* under the influence of gravity!

So what does this actually mean?

You will utilize a `javafx.animation.Timeline` to update the *position* and *velocity* of your Doodle, as determined by the equations of motion in the handout and the provided *GRAVITY* constant.

Remember! Since the positive y-axis runs down on a computer screen, the *GRAVITY* constant will be *positive*.
Let us assume that your Doodle initially starts at rest, with it's position set to (0,0) and its velocity set to 0.

At the end of your KeyFrame, you need to calculate a new position and velocity for your doodle using the following equations:

\[
\begin{align*}
\text{newVelocity} &= \text{previousVelocity} + \text{GRAVITY} \times \text{DURATION} \\
\text{currentPosition} &= \text{previousPosition} + \text{newVelocity} \times \text{DURATION}
\end{align*}
\]

where \( \text{previousPosition} = 0, \text{previousVelocity} = 0, \) \( \text{GRAVITY} = 1000 \) and \( \text{DURATION} = 15/1000 \) in this example
Using the previous equations, we get:
\[
\text{currentVelocity} = 15 \\
\text{currentPosition} = 225/1000
\]
Now we update Doodle’s variables:
\[
\text{y-velocity} \text{ is set to } 15 \\
\text{y-position} \text{ is set to } (\text{int})(225/1000) = 0
\]
And that's it! Just store the new values for \text{y-position} and \text{y-velocity} and repeat the process!
Collisions

How do we make the Doodle jump when it hits a platform?

- By using the `boolean intersects(double x, double y, double width, double height)` method of the `Node` class, that's how!
  - This method returns `true` if the `Node` that calls the method is overlapping a point in the “box” of size `width`, `height` that is sitting at the location `x`, `y`. This will work well if the `x`, `y`, `width`, and `height` correspond to the `Platform` we’re checking for intersection with.

  *Remember:* A Shape is a subclass of Node!

- Is it enough to just check if your Doodle is intersecting a platform? Or should your Doodle rebound only under certain conditions?
Now that we know how to check if our Doodle is colliding with one platform, how can we check to see if it's colliding with any platform?

- By using a loop! (We’ll explain more in a few slides.)
- As you check all your platforms for a collision, you just have to set your Doodle's y-velocity to the `REBOUND VELOCITY` constant if a collision is detected!
How will you make your Doodle move right and left?
You will be using an `EventHandler` just like in Cartoon, but the `Event` type will be `javafx.scene.input.KeyEvent`.

They work pretty much the same way:
1. You decide **where** you will have your listener.
2. You create your **inner class** that implements the interface.
3. You **specialize** how your class **handles** Events.

What is the difference then?

*KeyEvents and KeyCodes*
Key Codes

- **KeyCode**s are enums that represent the keys on a keyboard.

- Calling the method `getCode()` on the `KeyEvent` passed into your `handle` method will return the `KeyCode`.

- You can use the `KeyCode` enums to check the value returned by the `getCode()` method. For example, to check for up arrow button:

  ```java
  public void handle(KeyEvent e) {
      if(e.getCode() == KeyCode.UP){ ... }
  }
  ```

**Note:** see the handout for more detailed information.
Arraylists and Platforms

- How many platforms will you have to create?
  As many as needed to fill the screen!
- Where will you store all of your platforms?
  `java.util.ArrayList`.
- `ArrayList` is a built-in Java class. You DO NOT need to make your own ArrayList class. Use JavaDocs to find methods this class has.
- When using an `ArrayList`, just like arrays, you need to specify the type of element your arraylist will store. For DoodleJump, you will be storing your platforms in the `ArrayList`. So you will specify the name of your platform class in the literal `< >` brackets!
**Arraylists**

**Question:** How will you go through **ALL** of the elements in your `ArrayList`?

**Answer:** Use loops!

Look back at the Arrays lecture for a review on using loops to iterate through a list of items.

**Hint:** `for-each` loops are very helpful, but you will not be able to `remove` items from your list during this type of loop.

- Trying to do so will cause `ConcurrentModificationExceptions`!
You must generate platforms for your character to hop on and your platforms should be generated “semi-randomly”!

Remember: It’s important that platforms are reachable for the character.
1. When the program starts, create enough platforms to cover the entire screen using a `while` loop.

2. When the Doodle jumps to a higher platform, create new platforms!
   ○ We’ll outline the `generatePlatform()` method (next slide) for generating one new platform.

3. When the Doodle is above the center of the panel, move all the platforms down and remove any offscreen platforms.
Platform Generation Pseudocode

generatePlatform() {
    create new Platform
    find min x & max x for the new Platform
    calculate a random x position between min x and max x
    find min y & max y for the new Platform
    calculate a random y position between min y and max y
    set the new platforms location to the calculated positions
}

Think: What are your constraints on \( \text{min } x \) and \( \text{max } x \)? How about \( \text{min } y \) and \( \text{max } y \)?
Random Numbers

How do you come up with a random location for the new Platform?

Math.random() returns a double value greater than or equal to 0.0 and less than 1.0

Example:

min = minimum reachable x location
max = maximum reachable x location
min + (int)((max - min + 1) * Math.random())

// gives a value between min and max inclusive
Vertical Scrolling

When the Doodle tries to pass a certain height, it should stop moving, and all the platforms move downward

- The Doodle’s y-coordinate should be moved to the midpoint.
- The platforms should move the remaining distance the Doodle would have moved, but in the positive Y direction (downward).
If doodle is above screen midpoint:
Calculate and store how far the Doodle would move up
Set the Doodle's position to the panel midpoint
For each platform:
    Lower by how much the Doodle would have moved
Start early … start today …

Start yesterday!