TopHat Question

[Anonymous]: How did you feel about the water skit last lecture and the raunchy elements of our skits in general?

A.  I don't want to see anything like that again; please tone it down
B.  I didn't really like it; I'd prefer if you tone it down, but I am generally indifferent
C.  I feel neutral
D.  I thought it was a little funny
E.  I thought it was very funny
Revisiting Arrays: Size of 2-D Arrays

public static final int NUM_ROWS = 10; // defined in Constants
public static final int NUM_COLS = 6; // defined in Constants

public void practice2DArrays() {
    // deciding which is row and which is column index is
    // arbitrary but must be consistent!!!
    String[][] stringArray = new String[NUM_ROWS][NUM_COLS];
    int numRows = stringArray.length;
    int numCols = stringArray[0].length;
    System.out.println("My array has " + numRows * numCols + " slots in total!");
}

array.length gives size of first dimension (you decide whether you want row or column), and array[0].length gives size of second dimension
Common Array Errors - Watch Out! (1/2)

- Cannot assign a scalar to an array

  \[
  \text{int}\[] \text{ intArray} = 5; \quad \times
  \]

  - 5 is not an array
    - to initialize array elements, must loop over array and assign values at each index. Here we assign 5 to each element:

  ```java
  int[] intArray = new int[20]; //initializes array, not elements
  for (int i=0; i < intArray.length; i++){
    intArray[i] = 5;
  }
  ```
Common Array Errors - Watch Out! (2/2)

- Cannot assign arrays of different dimensions to each other

```java
int[] intArray = new int[23];
int[][] 2DIntArray = new int[2][34];

X intArray = 2DIntArray;
```

- Doing so will result in this error:

  "Incompatible types: Can’t convert int[] to int[][]"

- Similar message for assigning arrays of mismatched type

- Take note that Java will automatically resize an array when assigning a smaller array to a larger one
2D Arrays Example (1/2)

• Let’s build a checkerboard with alternating black and white squares, using JavaFX
• Each square has a row and column index
• Let’s use row-major order
  o access any square with `checkerboard[rowIndex][colIndex]`
• JavaFX `Rectangle`’s location can be set using row and column indices, multiplied by square width factor
  o row indicates Y values, column indicates X value
// instantiate a Pane and initialize the checkboard 2D array
Pane myPane = new Pane();
Rectangle[][] checkerboard = new
    Rectangle[Constants.NUM_ROWS][Constants.NUM_COLS];
// loop through row and column indices (for each col in each row...)
for (int row = 0; row < checkerboard.length; row++) {
    for (int col = 0; col < checkerboard[0].length; col++) {
        // instantiate rectangle, setting Y/X loc using row/col indices
        Rectangle rect = new Rectangle(col * Constants.SQ_WIDTH,
            row * Constants.SQ_WIDTH,
            Constants.SQ_WIDTH,
            Constants.SQ_WIDTH);
        // alternate black and white colors
        if ((row + col) % 2 == 0) {
            rect.setFill(Color.BLACK);
        } else {
            rect.setFill(Color.WHITE);
        }
        myPane.getChildren().add(rect); // graphically add the rectangle
        checkerboard[row][col] = rect; // logically add the rectangle
    }
}
SciLi Tetris: Loops and Arrays Writ Large

• In 2000, Tech House constructed then the largest Tetris game on the Scili – the Woz flew out to play it!

• 5 months of work: 11 custom-built circuit boards, a 12-story data network, a Linux PC, a radio-frequency video game controller, and over 10,000 Christmas lights – see http://bastilleweb.techhouse.org/

• Video: https://www.youtube.com/watch?v=tklRWoo9qrU&t=21s

• Article: http://news.bbc.co.uk/2/hi/science/nature/718009.stm
Lecture 14

Design Patterns and Principles: Part 1

From xkcd Webcomics: https://xkcd.com/974/
“Design-Focused” Projects (1/2)

• Projects up to and including Fruit Ninja were considered “foundation-focused”

• Projects for remainder of semester are considered “design-focused”
  o given only an assignment specification (and hints), you will design programs from scratch

• On early projects, design was 25% of code grade; now 30-35%
  o for at least two of the following Fruit Ninja, Cartoon, Doodle Jump, Tetris, you will have Code Debriefs
  o 8% of your final grade is made up of Code Debriefs, where you will describe your design and code to TAs
“Design-Focused” Projects (2/2)

• Put much more effort (≥ 2-3 hours) into understanding assignment specifications and planning before coding
  - containment/association and interface/inheritance diagrams crucial!

• Starting to code with a poor design leads to hours wasted trying to design and code on the fly
Design Grading

• Cartoon design grade will be based on design guidelines in the handout and discussed throughout this semester
  o will NOT be graded on specifics of this lecture

• Remaining projects’ design WILL be graded with this week’s design patterns + principles lectures in mind
  o refer to this lecture when designing DoodleJump with your partner!
Outline

- Design in a Nutshell
- Abstraction and Encapsulation
- Class Cohesion and Coupling
- Wrapper Classes
Context Beyond CS15

• Imagine you’re working for a company with a bunch of software engineers that write the code for a popular app

• The app needs to work properly now, and in the future, more engineers will need to add new/change existing features

• Your job is to write code that:
  o works properly (*functionality*)
  o is easily readable (*style*)
  o another engineer can add to easily (*design*)
  o another engineer can modify easily (*design*)

• When writing real code, the *design* of your program is ultimately as important as its functionality
Design in a Nutshell (1/2)

● Up to now, focused on how to program
  ○ be appropriately lazy: re-use code and ideas
● Increasingly we learn about **good design**
● Some designs are better than others
  ○ “better” means, for example:
    ▪ more efficient in space or time required (traditional criteria)
    ▪ more robust, the “ilities” – usability, maintainability, extensibility, scalability…
● These are central concerns of **Software Engineering**
  ○ discussed in detail in CS32 (CSCI0320)
Design in a Nutshell (2/2)

- There are trade-offs to make everywhere
  - architect balances aesthetics, functionality, cost
  - mechanical engineer balances manufacturability, strength, maintainability, cost

- Need to defend your trade-offs
  - no perfect solution, no exact rules
  - up to now designs rather straight-forward, not concerned about performance because not dealing with larger collections of data
What Do We Cover in These Lectures?

• Walk through process of planning design for a mock CS15 project

• Emphasize design principles and design patterns, which will be directly relevant to projects (including DoodleJump!),
Our Mock CS15 Project: Snake!

• Snake moves around board of squares at specified rate and continues moving in its last direction

• Player changes snake direction via key input, with goal of eating pellets to increase score

• Snake starts 3 squares in length, grows 1 square for each pellet eaten

• Snake can only move forward and turn right or left relative to its direction, not 180°

• Gain score by eating pellets – different colors yield different scores

• Game ends when snake moves off screen or into itself
Outline

- Design in a Nutshell
- Abstraction and Encapsulation
- Class Cohesion and Coupling
- Wrapper Classes
Where do I start?!

• Assignment specifications can be daunting

• Start at highest level: brainstorm how to separate components of program (delegation pattern!)
  o containment/association decisions
  o what classes should we write? how should they communicate with each other?
  o critical to consider where to divide abstractions
Recall: Delegation Leads to Abstraction

- Delegation results in **levels of abstraction**, where each level deals with more specifics to complete an action

- Please groom my dog!
- Wash this dog with shampoo, then trim its hair and dry!
- Fill the bath with warm water until it’s two-thirds full…

- DogOwner
- PetShop
- DogGroomer

- Bath
- HairDryer
- Clippers
Abstractions (1/3)

- Each class represents an abstraction
  - a “black box”: hides details that external users do not care about
  - allows you as the programmer to control programs’ complexity – only think about relevant features
Abstractions (2/3)

- CS15 support code and JavaFX are great examples of levels of abstraction.

```java
andyBot.turnRight();
this.pongBall.updatePosition();
vbox.setAlignment(Pos.TOP_CENTER);
```

- Don't need to worry about internals of how robot moves.
- Ball tracks its own speed and direction – we just tell it to move.
- JavaFX handles pixel-level graphical representation and display.
Abstractions (3/3)

• CS15 support code itself results in **levels of abstraction**
  o each layer becomes more specific

```
this.pongBall.updatePosition();
```

```
circle.setCenterY(circle.getCenterY() + verticalChange);
```

JavaFX internals to manipulate specific pixels
Encapsulation(1/2)

• Lack of clean abstractions leads to messy communication between classes

• Example: Game class contains CompositeShape class that moves across screen
  ○ must allow access to CompositeShape’s private components via getHead() and getBody()
  ○ each communication between Game and CompositeShape internals is an arrow connecting them

• With access to those shapes, Game could also write code like this.getHead().setFill(Color.RED);
  ○ but what if we don’t want Game to be allowed to change CompositeShape’s color?!
Encapsulation(2/2)

• We do this by…
  
  o delegating details to `CompositeShape`, simplifying communication
  
  o abstracting details of moving shapes, means no more need for `getHead()` and `getBody()`
  
  o so, `Game` doesn’t need to know the details of moving shapes!

• Clean abstractions leads to clear communication between classes

• **Key Point:** Use **getters/setters** ONLY as necessary to maximize encapsulation safety (IMPORTANT for future courses like CS200)

```java
public void moveRight()
{
  this.head.setX(20 + this.head.getX());
  this.body.setX(20 + this.body.getX());
}
```
Outline

- Design in a Nutshell
- Abstraction and Encapsulation
- Class Cohesion and Coupling
- Wrapper Classes
Review: Composition Pattern (1/3)

- You’ve used composition from the beginning

- Models object built through its containment of other objects and/or association with peer objects

- This is a has-a relationship, in which an object has an instance of another class stored as an instance variable
  - can be modeled through both containment – using the new keyword
  - as well as association – passing an object to an instance of another class to store as one of its instance variables

- Think of instance variables as modeling both the components and the properties/attributes that make up a class
Review: Composition Pattern (2/3)

- Compose one object out of other, more specialized objects that do one specific thing, e.g., car’s engine
  - factor out code that works together for one specific purpose into a separate class (ex. `heatOven()` & `bakeCookies()` can go into a `Baker` class)
    - only instantiate an instance of this class if you need that functionality
  - specialist classes allow you to design components that you can build on
    - i.e., black boxes that expose only limited functionality
    - this is a form of delegation – don’t rewrite code that specialists can do for you!

- Think of these specialist classes like Lego blocks that you can piece together to compose a larger class
  - every type of Lego block is unique and serves a specific purpose in your overall structure
Review: Composition Pattern (3/3)

- How can we determine good delegation and composition decisions?
- A Car class would use instances of these classes
  - Engine, Brake, Transmission, SeatBelt, ...
  - Car can delegate startUp() to the Engine, ...
High Cohesion and Loose Coupling (1/3)

• Cohesion refers to how well-defined the purpose of a single class is

• A class with a single, well-defined purpose has high cohesion
  
  o This is also known as the Single Responsibility Principle

• Strong separation of concerns reduces mental juggling – when coding in one class, only need to think about limited pieces of functionality – avoid “Swiss army knife” classes!

• You should be able to succinctly describe the purpose of each class in class header comments
## High Cohesion or Low Cohesion?

<table>
<thead>
<tr>
<th>High Cohesion</th>
<th>Low Cohesion</th>
</tr>
</thead>
<tbody>
<tr>
<td>• In a program modeling the life of a student, there is one <strong>CS15</strong> class for the student to track their CS15 assignments</td>
<td>• In a program modeling the life of a student, there is one <strong>Life</strong> class that handles Fall classes, social life, and extracurriculars</td>
</tr>
<tr>
<td>• In Cartoon, one class that models a <strong>Cloud</strong> with 5 circles and moves each of the circles across the pane</td>
<td>• In Cartoon, <strong>PaneOrganizer</strong> handles setting up the overall structure of panes, subpanes and shapes, and handles changing the color of each shape on key presses</td>
</tr>
</tbody>
</table>
High Cohesion and *Loose Coupling* (2/3)

- Coupling refers to how interdependent two classes are
- Each class should have *loose coupling* with other classes
  - use *abstractions* to keep clear relationships between classes
- **Limit dependencies between classes**
  - should be able to modify internals of one class without worrying about impact on other classes
Coupling Example (1/3)

• Back to shape movement! Let’s say we have our app to make a planet move via `Planet` class
  o to start, the planet is just represented by a `Circle`

```java
public class Planet {
    private Circle circle;
    public Planet() {
        this.circle = new Circle(Constants.PLANET_RADIUS);
    }
    public Circle getCircle() { return this.circle; }
}

// in Cartoon class
Planet venus = new Planet();
Timeline timeline = new Timeline(Duration.seconds(1), (ActionEvent e) ->
    venus.getCircle().setX(venus.getCircle().getX() + 10));
```
Coupling Example (2/3)

• Now we decide to use a composite shape with 4 rings around the planet
• First, move the Circle from Planet class
• Then, move the rings from Planet class
• Now every time a shape is added, it must be moved in Cartoon
• This is tight coupling (bad), i.e., Cartoon is too involved with details of moving Planet

```java
// in Cartoon class
Planet venus = new Planet();
Timeline timeline = new Timeline(Duration.seconds(1), (ActionEvent e) -> {
    venus.getCircle().setX(venus.getCircle().getX() + 10);
    venus.getRing1().setX(venus.getRing1().getX() + 10);
    venus.getRing2().setX(venus.getRing2().getX() + 10);
    // etc.
});
```
Coupling Example (3/3)

- Alternatively, could just have one `move` method in `Planet`
- `Planet` could have 1 shape or 18 shapes, and `Cartoon` doesn’t need to change!
- This is **loose coupling** (good)

```java
public class Planet {
    // constructor and instance variables elided
    public void move() {
        // This method could move one shape or a bunch of shapes, but
        // Cartoon doesn’t need to know about the details!
    }
}

// in Cartoon class
Planet venus = new Planet();
Timeline timeline = new Timeline(Duration.seconds(1),
    (ActionEvent e) -> venus.move());
```
High Cohesion and Loose Coupling (3/3)

• **Key Point:** Each class should have an independent, well-defined purpose (*high cohesion*), and communication between classes should be as simple and well-defined as possible (*loose coupling*)
A *Tribute* class is using an instance of the *Bow* class to hunt for food. Which code in *Tribute* would indicate that the *Bow* class is written with proper encapsulation, abstractions, and loose coupling?

A. `bow.getQuiver().getArrow().shoot();`

B. `bow.shootArrow();`

C. `bow.nockArrow("Wooden");
   bow.drawBowString();
   bow.looseArrow("Wooden");`
Back to Snake Brainstorming (1/3)

• Start at highest level: brainstorm how to separate components of program
  o keeping in mind aim of **high cohesion** and **loose coupling**

- **PaneOrganizer** — Organize high-level graphical elements of game
- **SnakeGame** — Handle high-level logic of game through timeline and key input
- Data structure to represent a board...
  - 2D array of squares
- Data structure to represent snake...
  - **ArrayList** because length changes

**We succinctly described purpose of these components – indicator of high cohesion!**
C/A Diagram Draft

App

PaneOrganizer

SnakeGame

Rectangle
• Let’s think more about what’s going on in the SnakeGame class
• What should happen at each tick of the Timeline?
• Let’s write pseudocode:
  o move snake into next tile
  o if snake went off board or ran into itself:
    ▪ game over
  o if pellet is on tile that snake moved into:
    ▪ eat pellet
    ▪ add to score
    ▪ Increase snake length by one square
    ▪ generate new pellet
Back to Snake Brainstorming (3/3)

• We realize that each board square needs some extra information
  o is snake on the square?
  o is pellet on the square?

• With more complexity, let’s consider delegating to a class `BoardSquare` rather than making `SnakeGame` handle it
  o instead of a board of “simple squares” (javafx `Rectangles`), we need “smart squares” (our own `BoardSquare` class)
  o then we can model this extra information as properties (instance variables) of the square!
Designing the **BoardSquare** (1/3)

- Since each `BoardSquare` represents one graphical square, should we have `BoardSquare` inherit from a JavaFX `Rectangle`? Similar to a sports car inheriting from a car…

```java
public class BoardSquare extends Rectangle {
    // ...
}
```
Designing the **BoardSquare** (2/3)

- If `BoardSquare` extends `Rectangle`, `BoardSquare` inherits all of `Rectangle`’s methods.

- That means `BoardSquare`’s set of public methods becomes the `Rectangle`’s set of public methods *plus* whatever specialized methods we write: is that a feature or a bug?

- In the context of Snake, we don’t want programmers to have access to all `Rectangle` methods -- if they did, they could change position, size, rotation, etc. of `BoardSquare`
Designing the **BoardSquare** (3/3)

- **Key point:** To achieve simple communication between classes (**loose coupling**), the set of public methods a class or interface exposes should be as **simple** and **restricted** as possible.

- Remember **encapsulation**... keep private parts your own business.

- Let’s only allow users of **BoardSquare** to access the limited parts we *need* to make public.

- In this case, most of **Rectangle** methods shouldn’t be accessible – how can we make them private?
Outline

- Design in a Nutshell
- Abstraction and Encapsulation
- Class Cohesion and Coupling
- Wrapper Classes
Wrapper Classes

• A wrapper is code that encapsulates (or “wraps” around) another software component as a layer of abstraction

• In Java specifically, we create wrapper classes that add a layer of abstraction to another Java class
  ○ i.e., we add functionality to a class that other classes using it do not need to know details of

• Instead of inheriting from a class, contain an instance of that class as a component (in an instance variable)
**BoardSquare** Wrapper Class (1/2)

- **BoardSquare** wraps an instance of **Rectangle**
  - **Rectangle** is the main component of **BoardSquare**, but it also has extra functionality/information

- Allows us to restrict certain accesses inherited from **Rectangle** and add helpful pieces of information
  - **Pellet** contained in a **BoardSquare**
  - **original Color** of a **BoardSquare**

```java
public class BoardSquare {
    private Rectangle square;
    private Pellet pellet;
    private Color originalColor;

    public BoardSquare(Pane pane, boolean odd) {
        this.square = new Rectangle();
        this.pellet = null;
        if (odd) {
            this.originalColor = Color.GREEN;
        } else {
            this.originalColor = Color.YELLOW;
        }
        this.setUpSquare(); // set size, location, ...
        pane.getChildren().add(this.square);
    }
}
```
BoardSquare Wrapper Class (2/2)

• A wrapper class exposes just the info that needs to be public and no more!
  ○ generally via setter and getter methods

• To show snake moving across board, one way is to change color of square to Black
  ○ so we add a setter for Color

```java
public class BoardSquare {
    private Rectangle square;
    private Pellet pellet;
    private Color originalColor;

    public BoardSquare(Pane pane, Boolean odd) {
        // constructor body elided
    }

    public void setColor(Color color) {
        this.square.setFill(color);
    }
}
```
Keep Class Relationships Simple! (1/2)

• Is `setColor` the best we can do to abstract away internals of the square?

• For our game, we want:
  o square to turn black when snake goes over it
  o square to return to original color when snake moves off it

• With `setColor`, programmer could make square any arbitrary color – that shouldn’t happen!

```java
public class BoardSquare {
    private Rectangle square;
    private Pellet pellet;
    private Color originalColor;

    public BoardSquare(Pane pane, Boolean odd) {
        // constructor body elided
    }

    public void setColor(Color color) {
        this.square.setFill(color);
    }
}
```
Keep Class Relationships Simple! (2/2)

• Instead, let’s have two separate methods
  
  o one method for snake moving onto square
  
  o one method for snake leaving square

• Trade-off: this produces more code but makes relationship between classes simpler (looser coupling)

• Key Point: Strive for simpler class relationships – that may not always mean fewer methods!

```java
public class BoardSquare {
    private Rectangle square;
    private Pellet pellet;
    private Color originalColor;

    public BoardSquare(Pane pane, Boolean odd) {
        // constructor body elided
    }

    public void addSnake() {
        this.square.setFill(Color.BLACK);
    }

    public void reset() {
        this.square.setFill(this.originalColor);
    }
}
```
Containment over Inheritance

• Wrapper classes are a good example of a generally agreed-upon design principle that containment is preferred to inheritance, unless the inheriting class should publicly expose all methods inherited.

• In our Snake example, our wrapper class is designed so BoardSquare has-a Rectangle as opposed to BoardSquare is-a Rectangle.
TopHat Question

Which of the following is NOT true about wrapper classes?

A. The goal of a wrapper class is to make a class’s set of public methods as simple as possible
B. Wrapper classes are an example of using encapsulation
C. Wrapper classes add a layer of abstraction around some contained class
D. Wrapper classes use inheritance rather than composition
Representing the Snake (1/2)

• Let’s consider how to use **ArrayList** to represent the snake

• What should the **ArrayList** hold?
  - **BoardSquares** – hold whichever squares that snake is on top of
  - type will be **ArrayList<BoardSquare>**
Representing the Snake (2/2)

• **ArrayList** could be an instance variable in **SnakeGame** class… or could delegate it!
  o delegate for **higher cohesion**
• **Snake** class will act as **wrapper** class for **ArrayList<BoardSquare>** and **only** expose method to **move** and **changeDirection**
  o so much simpler than including all **Rectangle** methods
• **Important note**: This decision means **SnakeGame** class won’t have direct access to **ArrayList** so it can’t mess with contents of list directly (**encapsulation**)
Representing the Board

• We model our static board with a 2D array \( \text{BoardSquare}[][] \)
• Once board is created, the only editing to it will be to change state of individual \( \text{BoardSquare} \)
• Delegate to a \( \text{Board} \) class that acts as wrapper of \( \text{BoardSquare}[][] \)?
  o definitely \textcolor{red}{high cohesion} since \( \text{Board} \) would only handle board contents
  o no major benefit of delegating to \( \text{Board} \) as a wrapper since likely the only method would be a getter
    ▪ \textcolor{red}{public BoardSquare tileAt(int row, int col) }
  o could argue for or against having separate \( \text{Board} \) class – both solutions are on GitHub!
## Recap of Design Brainstorming So Far (1/2)

<table>
<thead>
<tr>
<th>Class</th>
<th>Purpose</th>
<th>Important Instance Variables</th>
<th>Important Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>App</td>
<td>Starts the application</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>PaneOrganizer</td>
<td>Organizes the high-level graphical organization of the program</td>
<td>BorderPane root</td>
<td>n/a</td>
</tr>
<tr>
<td>SnakeGame</td>
<td>Handles high-level logic of game via timeline and key input</td>
<td>Pane gamePane, Snake snake, Board board</td>
<td>updateGame (called on timeline), handleKeyInput (called on key press)</td>
</tr>
</tbody>
</table>
## Recap of Design Brainstorming So Far (2/2)

<table>
<thead>
<tr>
<th>Class</th>
<th>Purpose</th>
<th>Important Instance Variables</th>
<th>Important Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>Snake</td>
<td>Represents snake moving snake moving around the board</td>
<td>ArrayList&lt;BoardSquare&gt; snakeSquares, Board myBoard, // to store association Direction currDir // Direction enum – Thu’s lecture!</td>
<td>move, changeDirection</td>
</tr>
<tr>
<td>Board</td>
<td>Represents board of squares</td>
<td>BoardSquare[][] board</td>
<td>tileAt</td>
</tr>
<tr>
<td>BoardSquare</td>
<td>Represents one square on the board</td>
<td>Rectangle boardSquare, Color originalColor, Pellet pellet</td>
<td>addSnake, reset, isEmpty</td>
</tr>
</tbody>
</table>
Containment/Association Diagram

App

PaneOrganizer

SnakeGame

Board

via 2D array

Snake

via ArrayList

BoardSquare
Class Diagram

Key

Class
- instance variables
+ methods

App

PaneOrganizer
- root : BorderPane

SnakeGame
+ updateGame() : void
+ handleKeyInput(KeyEvent e) : void

Board
- board : BoardSquare[][]
+ tileAt(int row, int col) : BoardSquare

BoardSquare
- square : Rectangle
+ addSnake() : void
+ reset() : void
+ isEmpty() : boolean

Snake
- snakeSquares : ArrayList<BoardSquare>
- currDirection : Direction
+ move() : BoardSquare
+ changeDirection(Direction dir) : void

Note: There is subtlety here around the BoardSquare. Board creates BoardSquares, and Snake gets BoardSquare instances from Board to store in its ArrayList; Snake does not contain its own BoardSquares, it only stores references to BoardSquares already instantiated by Board.
Announcements (1/2)

• **Snake code on GitHub** – check it out to see contrasting design decisions, and example of large program implementation
  o don’t worry if some of it doesn’t make sense, we will continue during Thursday’s lecture

• 1D Arrays, ArrayLists, and Loops Section this week!
  o be sure to complete **mini-assignment** and send to section TAs prior to section

• DoodleJump Released Today!!
  o early handin: Monday 10/30
  o on-time handin: Wednesday 11/1
  o late handin: Friday 11/3
  o do not underestimate this assignment! start early!
Announcements (2/2)

• Code-Along: Debugging and GitHub
  o Wednesday October 25\textsuperscript{th}
  o Sunday October 29\textsuperscript{th}
DoodleJump: Getting Started

• What classes should you represent in DoodleJump? What should their containment/association relationships be?

• How can you leverage “wrapper classes” to wrap some JavaFX elements you use to represent components of the program?

• How can you model properties like game score and doodle velocity? Which classes are those properties of?

• What do the different platforms have in common, and how are they different? How can you leverage polymorphism to make it so that the game doesn’t need to know the actual type of each platform it moves?
Software Development: A 5-Step Process

- Analysis
- Design
- Implementation
- Testing
- Maintenance
Software Development: A 5-Step Process

**Analysis** has been done for us via assignment specification.

**Design** is where we’re focusing today!

**Implementation** is when you code!

**Testing**, in CS15, typically means playing your game.

**Maintenance** isn’t as applicable in CS15.
Extra Credit Discussion Results!

- Should targeted advertising be allowed under the ADPPA?
  - Yes: 56.7%
  - No: 43.3%

- Should whistleblowers be protected under the ADPPA?
  - Yes: 81.8%
  - No: 18.2%

- Should the ADPPA override state privacy laws?
  - Yes: 45.3%
  - No: 54.7%
Surveillance Capitalism

CS15 Fall 2023
Surveillance Capitalism

“I describe surveillance capitalism as the unilateral **claiming of private human experience as free raw material for translation into behavioral data.** These data are then computed and packaged as prediction products and **sold into behavioral futures markets.**”

– Shoshana Zuboff (retired Harvard Business School Professor and author of *The Age of Surveillance Capitalism*), 2019

**Surveillance capitalism** is when companies gather **our private information**, **analyze** it, and then **sell insights** about our behavior **to other businesses.**
Industrial capitalism

Production → Sales → R&D → Profit
Origins of Surveillance Capitalism

Adapted from The Age of Surveillance Capitalism by Shoshana Zuboff. Image source: Adobe Stock.
Origins of Surveillance Capitalism

Adapted from The Age of Surveillance Capitalism by Shoshana Zuboff. Image source: Adobe Stock.
Origins of Surveillance Capitalism

User Input → Rendered Behavior → Data Exhaust
(Used to generate predictions on user’s education backgrounds, emotions…)
(… which help businesses with targeted advertising…)

Service Improvements

New customer!

How Companies Identify Users

Diagram adapted from Cracked Labs, 2017
Tracking You: Fingerprinting “Signals”

1. Platform (4)
2. Color Depth (5)
3. Screen Resolution (10)
4. CPU class (100)
5. Full list of installed fonts (maintaining their order, which increases the entropy) (5000)
6. Timezone
7. UserAgent
8. Language
9. Has session storage or not
10. Has local storage or not
11. Has indexed DB
12. Has IE specific ‘AddBehavior’
13. Has open DB

All numbers in parentheses above are approximations, and the animations assume an even distribution, which is of course not the case.
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Platform (4)</td>
</tr>
<tr>
<td>2.</td>
<td>Color Depth (5)</td>
</tr>
<tr>
<td>3.</td>
<td>Screen Resolution (10)</td>
</tr>
<tr>
<td>4.</td>
<td>CPU class (100)</td>
</tr>
<tr>
<td>5.</td>
<td>Full list of installed fonts (maintaining their order, which increases the entropy) (5000)</td>
</tr>
<tr>
<td>6.</td>
<td>Timezone</td>
</tr>
<tr>
<td>7.</td>
<td>UserAgent</td>
</tr>
<tr>
<td>8.</td>
<td>Language</td>
</tr>
<tr>
<td>9.</td>
<td>Has session storage or not</td>
</tr>
<tr>
<td>10.</td>
<td>Has local storage or not</td>
</tr>
<tr>
<td>11.</td>
<td>Has indexed DB</td>
</tr>
<tr>
<td>12.</td>
<td>Has IE specific ‘AddBehavior’</td>
</tr>
<tr>
<td>13.</td>
<td>Has open DB</td>
</tr>
</tbody>
</table>

All numbers in parentheses above are approximations, and the animations assume an even distribution, which is of course not the case.
1. Platform (4)
2. Color Depth (5)
3. Screen Resolution (10)
4. CPU class (100)
5. Full list of installed fonts (maintaining their order, which increases the entropy) (5000)
6. Timezone
7. UserAgent
8. Language
9. Has session storage or not
10. Has local storage or not
11. Has indexed DB
12. Has IE specific ‘AddBehavior’
13. Has open DB

All numbers in parentheses above are approximations, and the animations assume an even distribution, which is of course not the case.
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Platform (4)</td>
<td></td>
</tr>
<tr>
<td>Color Depth (5)</td>
<td></td>
</tr>
<tr>
<td>Screen Resolution (10)</td>
<td></td>
</tr>
<tr>
<td>CPU class (100)</td>
<td></td>
</tr>
<tr>
<td>Full list of installed fonts (maintaining their order, which increases the entropy) (5000)</td>
<td></td>
</tr>
<tr>
<td>Timezone</td>
<td></td>
</tr>
<tr>
<td>User Agent</td>
<td></td>
</tr>
<tr>
<td>Language</td>
<td></td>
</tr>
<tr>
<td>Has session storage or not</td>
<td></td>
</tr>
<tr>
<td>Has local storage or not</td>
<td></td>
</tr>
<tr>
<td>Has indexed DB</td>
<td></td>
</tr>
<tr>
<td>Has IE specific ‘AddBehavior’</td>
<td></td>
</tr>
<tr>
<td>Has open DB</td>
<td></td>
</tr>
</tbody>
</table>

All numbers in parentheses above are approximations, and the animations assume an even distribution, which is of course not the case.
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>6. Timezone</td>
<td></td>
</tr>
<tr>
<td>7. UserAgent</td>
<td></td>
</tr>
<tr>
<td>8. Language</td>
<td></td>
</tr>
<tr>
<td>9. Has session storage or not</td>
<td></td>
</tr>
<tr>
<td>10. Has local storage or not</td>
<td></td>
</tr>
<tr>
<td>11. Has indexed DB</td>
<td></td>
</tr>
<tr>
<td>12. Has IE specific ‘AddBehavior’</td>
<td></td>
</tr>
<tr>
<td>13. Has open DB</td>
<td></td>
</tr>
</tbody>
</table>

All numbers in parentheses above are approximations, and the animations assume an even distribution, which is of course not the case.
<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Platform (4)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2. Color Depth (5)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3. Screen Resolution (10)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4. CPU class (100)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5. Full list of installed fonts (maintaining their order, which increases the entropy) (5000)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6. Timezone</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7. UserAgent</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8. Language</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9. Has session storage or not</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10. Has local storage or not</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11. Has indexed DB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12. Has IE specific ‘AddBehavior’</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13. Has open DB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

All numbers in parentheses above are approximations, and the animations assume an even distribution, which is of course not the case.
Incentive one: gather “better” data
Incentive two: gather more data

More time users spend…

…more data collected…

…more accurate predictions and hence profit

Incentive for **addictive design** and **sensationalist content**
So what if predictions are accurate?

Source: Texas A&M University (2019)
‘You Are the Product’: Targeted by Cambridge Analytica on Facebook

I AM SOMEONE WHO...

<table>
<thead>
<tr>
<th></th>
<th>Disagree Strongly</th>
<th>Disagree a Little</th>
<th>Neither Agree nor Disagree</th>
<th>Agree a Little</th>
<th>Agree Strongly</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seeks adventure.</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
</tr>
<tr>
<td>Trusts others.</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
</tr>
</tbody>
</table>

What happens now?

The Facebook button is disappearing from websites as consumers demand better privacy

FTC Sues Kochava for Selling Data that Tracks People at Reproductive Health Clinics, Places of Worship, and Other Sensitive Locations

The Slow Death of Surveillance Capitalism Has Begun

A European Union ruling against Meta marks the beginning of the end of targeted ads.