Overview

- Enums
- Factory Pattern
- Testing
- Interfaces v. Inheritance
- Method Overloading

Back to Our Snake Program

- Specifications
  - player moves snake via key input around board of squares with goal of eating pellets to increase score
  - snake can pivot left or right but not 180º
  - gain score by eating pellets – different colors yield different scores
- Represent snake as ArrayList of BoardSquares and delegate to a wrapper Snake class
- Represent board as 2D array of BoardSquares and delegate to a wrapper Board class
- Today, we’ll cover details about snake movement and food

Significant revisions to Snake code by former HTAs Adam, Brandon, and Naaf!
Overview

- **Enums**
- **Factory Pattern**
- **Testing**
- **Interfaces v. Inheritance**
- **Method Overloading**

Snake Movement (1/3)

- Snake keeps moving in the same direction until a key is pressed, which triggers change in direction
- Direction in which the snake moves is a **property** or piece of **state**
- What have we learnt so far that we can use to represent a **property** or piece of **state** for a class?
  - instance variables!
- Need to indicate whether direction the snake is moving is up, down, left, or right
- What type should our instance variable be?

Snake Movement (2/3)

- Can use **Strings** to store current direction of snake movement
  - Pros: easy to read and understand
  - Cons: value of strings can be anything!
    - e.g., the north direction can be represented as "up", "upward", "UP", "upside" and many more
    - can be confusing, it’s easy to mistype a string causing runtime bugs

```java
public class Snake {
    private String currDirection;

    public Snake() {
        this.currDirection = "up";
    }
}
```
Snake Movement (3/3)

- Alternatively, use integers to store current direction of snake movement.
  - Pros: It is less likely to mistype an integer compared to a string.
  - Cons: The numbers used are arbitrary:
    - e.g., 1 can mean anything. If 1 is up, is down -1 or 2?
    - Somebody reading your code wouldn't immediately understand what these numbers mean.
  - Neither of the choices so far are good enough.
  - Can think of directions as constants e.g., the cardinal points of a compass:
    - Need an easier way to store current direction from a set of constants.

```java
public class Snake {
    private int currDirection;
    public Snake() {
        this.currDirection = 1;
    }
}
```

Introducing Enums

- Enums are a special data type used to represent a group of related constants:
  - e.g., the cardinal directions: north, south, east, west.
  - Can create a `Direction` enum for this (next slide).
- The value of an `enum` can be any of the constants pre-defined for it:
  - The value of the `Direction` enum would be any of 4 directions.
- In our program, use enums to represent the cardinal directions of snake movement.

```java
public enum Direction {
    UP, DOWN, LEFT, RIGHT;
}
```

Declaring and Defining an Enum

- Declare it as `enum` rather than class or interface.
- Declare the set of constants, in this case the 4 directions, separating them with commas.
- Because they are constants, enum fields should be in all UPPER_CASE letters.
- To access the enum constants, use the dot syntax:
  - `Direction.up = Direction.UP;`
- Enums, just like classes, have their own `.java` file.
  - This file would be `Direction.java`.
  - In IntelliJ, an `enum` file will be shown by the letter `E`.
Using Enums: Snake Movement (1/3)

- Can use a `Direction` enum in `Snake` to store direction of movement
  - notice `currDirection`'s type is the enum `Direction`. Not `String` or `int`
  - `currDirection` is initialized to `right`
- Like any type in Java, enums can be used as parameters to methods
  - `changeDirection` sets current direction to whatever is passed in
- Notice how intuitive the value of `currDirection` is compared to when we used strings and integers!

```java
public class Snake {
    private Direction currDirection;
    public Snake() {
        this.currDirection = Direction.RIGHT;
    }
    public void changeDirection(Direction newDir) {
        this.currDirection = newDir;
    }
}
```

TopHat Question

Given the enum below, which of the following is a correct way to initialize the `paused` variable?

```java
public enum Status {
    PAUSED, RUNNING, STOPPED;
}
```

A. Status paused = new Status(PAUSED);
B. Status paused = Status(PAUSED);
C. Status paused = Status.PAUSED;
D. Status paused = Status.PAUSED;

Using Enums: Snake Movement (2/3)

- Remember the `handleKeyPress` method from lab4 & Cartoon?
  - JavaFX provided it with arguments that corresponded to Left, Right, or Space keys
  - these `KeyCode`s were ENUMs!
- Again, use a `switch` and call `changeDirection` in each case, passing in the corresponding direction
- But wait! There’s one specification with snake movement we’ve ignored
  - snake can pivot right or left, but not 180º
  - thus check new direction passed from key input is not the opposite of current direction

```java
private void handleKeyPress(KeyCode code) {
    switch (code) {
        case UP:
            this.snake.changeDirection(Direction.UP);
            break;
        case DOWN:
            this.snake.changeDirection(Direction.DOWN);
            break;
        case LEFT:
            this.snake.changeDirection(Direction.LEFT);
            break;
        case RIGHT:
            this.snake.changeDirection(Direction.RIGHT);
            break;
        case SPACE:
            this.snake.changeDirection(Direction.DOWN);
            break;
        default:
            break;
    }
}
Using Enums: Snake Movement (3/3)

- Can use a series of if-else statements to check that newDir is not the direction opposite currDirection
- Results in complicated code; need a simpler solution
  - given a direction, can we find its opposite?
  - how can we have this functionality be part of the enum so that snake can use it?

public class Snake {
// other methods elided
public void changeDirection(Direction newDir) {
    if (newDir == Direction.UP && this.currDirection != Direction.DOWN) {
        this.currDirection = newDir;
    } else if (newDir == Direction.DOWN && this.currDirection != Direction.UP) {
        this.currDirection = newDir;
    } else if (newDir == Direction.LEFT && this.currDirection != Direction.RIGHT) {
        this.currDirection = newDir;
    } else if (newDir == Direction.RIGHT && this.currDirection != Direction.LEFT) {
        this.currDirection = newDir;
    }
}

Introducing Enum Methods (1/3)

- Enums in Java act like classes in that we can define methods and other instance variables within its body
  - not a class, no constructor because values already enumerated in the declaration
- Can add a method, opposite, in our enum, that returns the opposite direction of the current direction
- But need to know what current direction (initialized in Snake’s constructor) is
  - can pass it to opposite as a parameter.
  - Anything wrong with this?
    - repetitive since Snake would call currDirection.opposite(currDirection);

public enum Direction {
    UP, DOWN, LEFT, RIGHT;
    public Direction opposite( ) {
        switch (     ) {
            case UP: return DOWN;
            case DOWN: return UP;
            case LEFT: return RIGHT;
            case RIGHT: return LEFT;
        }
    }
}

Enum Methods (2/3)

- Can instead pass this to switch statement
  - i.e., the value of Direction we call opposite on: current.opposite();
  - related to other uses of this
- If current is Direction.LEFT, what would current.opposite() return?
  - Direction.RIGHT

public enum Direction { 
    UP, DOWN, LEFT, RIGHT;
    public Direction opposite() { 
        switch (     ) { 
            case UP: return DOWN; 
            case DOWN: return UP; 
            case LEFT: return RIGHT; 
            case RIGHT: return LEFT; 
        }
    }
}
Enum Methods (3/3)

- Back in Snake, can now check that direction passed in from key input is not the opposite of current direction
- Use the != comparator to compare two enum values
- Notice how much simpler our code looks compared to the tower of if-else statements?
- Adding methods to enums makes them more robust and a useful data type to have in a program

TopHat Question

Given the enum below, which of the following could be a method in `Operator`?

```java
public enum Operator {
    ADD, SUBTRACT, MULTIPLY, DIVIDE;
}

public int calc(int a, int b) {
    switch(a, b) {
        case ADD: return a + b;
        case SUBTRACT: return a - b;
        case MULTIPLY: return a * b;
        case DIVIDE: return a / b;
    }
}
```

A. B. C.

Overview

- Enums
- Factory Pattern
- Testing
- Interfaces, Inheritance
- Method Overloading
Representing the Food (1/3)

- Goal is to grow the Snake as much as possible without moving it off screen or into itself
- Snake grows by eating pellets which are located on random positions on the board
- In our version of the game, want to model different types of the pellets
  - each with a different color and yielding different scores
- How can we generate these distinct types of pellets?

Representing the Pellets (2/3)

- Can use interface and create different Pellet classes that implement it?
- However, in the version of Snake we're making, there's very little difference between Pellet types
  - only difference is color and score which are properties of the class! No difference in functionality (methods)
- Important to keep in mind project specifications when designing because they affect our design choices
  - only if there were different actions associated with each pellet, might we want to use an interface

Representing the Pellets (3/3)

- Can use inheritance and factor out common implementation to super class e.g., graphically remove pellets from board once eaten?
- But in our program, there is only method (eat()) in Pellet. No need for super classes and sub classes
  - like using a sledgehammer to crack a nut!
- Even if we extended functionalities of Pellet so that the class had more capabilities, may need to override methods which can be dangerous (see addendum at end of deck!)
- Any other option?
  - recall how we generated different types of Scarfs in the Math and Making Decisions lecture
  - want to do something similar with Pellets
Factory Pattern (1/2)

- Can use **Factory Pattern** to create one **Pellet** class and specify parameters to its constructor that will configure its type, i.e., **score** and **color**
  - allows us to instantiate different types of **Pellets** without caring about creation logic
  - a really useful pattern when creation logic is more complicated, e.g., if each type of **Pellet** had a different shape. Or even with **Tetris** pieces (coming up soon!)

Factory Pattern (2/2)

- Key features: a switch statement
  - in this case uses a random number generator
  - used on Fruit Ninja to generate fruits/bombs

```java
public void spawnFood() {
    // gets random empty tile on board where food will be added
    BoardSquare tile = this.getRandomEmptyTile();
    Food food;
    int rand = (int) (Math.random() * 3)
    switch (rand) {
        case 0:
            food = new Pellet(this.gamePane, Color.RED, 10, tile.getRow(), tile.getCol());
            break;
        case 1:
            food = new Pellet(this.gamePane, Color.BLUE, 30, tile.getRow(), tile.getCol());
            break;
        default:
            food = new Pellet(this.gamePane, Color.GREEN, 50, tile.getRow(), tile.getCol());
            break;
    }
    tile.addFood(food);
}
```

Overview

- **Enums**
- **Factory Pattern**
- **Testing**
- **Interfaces v. Inheritance**
- **Method Overloading**
Testing Our Program (1/2)

Original "Waterfall Model" of Software Development

Typically test multiple stages of development!

Testing involves checking that the actual behavior of a program matches its expected behavior (you've done this by playing your games!)

Testing Our Program (2/2)

- You already test your programs all the time – by playing them!
- As we scale in complexity, we can incrementally test logic of our program beyond playing the game
- Unit Testing is useful for verifying that specific parts of our program work (ex. a method)
  - A rocket scientist would want to check her calculations and simulate takeoff before launching!
- How could we test our snake program without even running it?
  - e.g., check individual methods, such as isEmpty() method
    - returns false when either pellet or snake is on tile
    - returns true if no pellet or snake is on tile
  - How to test our methods (along with printlns, stacktrace, and debugger!)?
    - Isolate, isolate, isolate the problem!

Introducing JUnit Testing

- A framework for writing and running tests
- JUnit allows individual methods and edge cases to be tested in a controlled environment, a test suite
  - what if you need to test the end condition of a game that takes 100 hours to complete?
  - what if a bug only happens once every 1000 tries? Can't manually simulate!
- Unit Testing in CS15 has the following pattern:
  - set up testing class
  - instantiate essential objects required to test method(s)
  - use assertion methods to validate a boolean expression
- Assertion methods are JUnit methods we use to test
  - assertTrue(boolean condition) will pass if the boolean expression inside is true
  - assertFalse(boolean condition) will pass if the boolean expression inside is false
- You will get set up with JUnit in next weeks lab!!
JUnit Testing: Naïve Example

- Trivial example: test the following code that adds two integers:
- What steps do we take to test?

```java
public class Calculator {
    // constructor elided
    public int add(int x, int y) {
        return x + y;
    }
}
```

```java
public class CalculatorTestingSuite {
    @Test
    public void testAddNumbers() {
        Calculator calc = new Calculator();
        assertTrue(calc.add(2, 2) == 4);
    }
}
```

How does IntelliJ help?

- Our tests from the last slide look like this in IntelliJ:

  - Can run individual tests or the whole class (the "testing suite") with green play buttons:
  - Pressing the top play button, gets us the following output in IntelliJ:

```
@Test tells compiler this is a unit test
```

JUnit Testing: Snake Example

How can we apply this framework to test our Snake code?

```java
public class SnakeTestingSuite {
    @Test
    public void testTileUpdates() {
        Pane gamePane = new Pane();
        Board board = new Board(gamePane);
        Pellet pellet = new Pellet(gamePane, Color.RED, Constants.SCORE, 1, 1);
        BoardSquare tile = board.tileAt(1, 1);
        tile.addPellet(pellet);
        assertFalse(board.tileAt(1, 1).isEmpty());
        tile.addSnake(); // eats pellet, but adds snake
        assertFalse(board.tileAt(1, 1).isEmpty());
        tile.reset(); // removes snake
        assertTrue(board.tileAt(1, 1).isEmpty());
    }
}
```

How can we apply this framework to test our Snake code?

- Set up testing class
- Instantiate essential objects required to test method(s)
- Use assertion functions to validate boolean expression(s)

You will get practice writing tests like this in next weeks lab:

- Testing will be required for a mini-project alongside Tetris (after you learn more in lab) but not Doodle Jump
- You will learn a lot more about testing in CS200!!
Recap Snake Design Process

• Assignment specifications can be daunting
• Start at very high level: how to separate components of the program
  o which classes can I use to model different objects in my program?
  o what functionalities can I delegate to those classes?
  o how would those classes interact with each other?
  o how can you test these components?
  o is my design scalable?
  o repeat and revise!

Intermission

• Have seen how to design mock CS15 project from scratch
  o need to go through similar design discussions for the projects in the remainder of the semester
  o code for the different designs of Snake can be found on GitHub
• For remainder of lecture, will cover additional discussions around design that will be useful in the future

Overview

- Enums
- Factory Pattern
- Testing
- Interfaces, Inheritance
- Method Overloading
Interfaces vs. Inheritance

- When deciding between interfaces and inheritance, need to consider trade-offs between the two
  - interfaces + composition/containment offer more flexibility compared to inheritance
    - ex. wrapper classes, like a "smart square"
  - can implement several interfaces but only extend one super-class
  - while inheritance allows sub-classes to inherit functionality from parent, there's risk of unintended consequences when overriding methods
- Note that while interface (coupled with composition) is often favored over inheritance, there are use cases which can really take advantage of inheritance, e.g., cars and animals

Case 1: Problems with Inheritance

- Let's return to our Race example from the Inheritance lecture
- CS15Mobile, Van, and Convertible have many identical capabilities and share a lot of the same components
- start/stop engine
- We created a Car superclass
  - Car contains instances of Engine, Brake, etc.
- CS15Mobile, Van, and Convertible extend from Car

Extending Our Design

- Assume now that we add an ElectricCar class to the program
  - but ElectricCar does not use the standard Engine inherited from Car
  - can ElectricCar just override Car's methods that make use of Engine? Anything wrong with that?
    - can do this but could be dangerous (see appendix)
    - when you subclass Car, its this.engine, is hidden from you
    - a parent's private variables stay private
    - you inherit methods that use this.engine, but implementation is hidden
      - you do not know which methods use this.engine, let alone how they do that
    - and you still have the now useless this.engine via pseudo-inheritance
Case 2: Inheritance vs. Interfaces + Composition

- But how, if at all, are interfaces with composition any better?
  - Let’s consider the case below where we want to animate a clock

```java
public class AnimateClock {
    private Clock myClock;

    public AnimateClock(Clock c) {
        this.myClock = c;
        this.setUpTimeline();
    }

    private void setUpTimeline() {
        KeyFrame kf = new KeyFrame(Duration.seconds(1),
            (ActionEvent e) ->
                this.clock.tick());
        // code to add kf to timeline and start timeline
    }
}
```

Inheritance vs. Interfaces + Composition

- Will both of these solutions work if we pass in a GrandfatherClock object to AnimateClock(...) in the previous slide? GrandfatherClock only adds a Ding

```java
public class Clock {
    public Clock () {
        // code elided
    }

    public void tick() {
        /* code to update time, including delegation to HourHand's and MinuteHand's move() methods */
    }
}
```

```java
public class GrandfatherClock extends Clock {
    public GrandfatherClock () {
        // code elided
    }

    @Override
    public void tick() {
        super.tick();
        if (this.isEvenHour()) {
            this.playDing();
        }
    }
}
```

```java
public interface Clock {
    public void tick();
}
```

```java
public class GrandfatherClock implements Clock {
    private HourHand hourHand;
    private MinuteHand minuteHand;

    public GrandfatherClock() {
        // instantiate HourHand and MinuteHand
    }

    @Override
    public void tick() {
        this.minuteHand.move();
        this.hourHand.move();
        if (this.isEvenHour()) {
            this.playDing();
        }
    }
}
```

Different Implementations, Same Result

- Both of these implementations result in a GrandfatherClock animating correctly
  - In solution 1, Clock is a superclass
  - In solution 2, Clock is an interface
  - Both can be used polymorphically

- But pros and cons to each solution
Inheritance Design: Pros and Cons

**Pros:**
- Better code reuse
  - Methods are automatically inherited in subclasses, so no need to re-implement functionality. For example, `tick()` delegate most of the responsibility to a `MinuteHand` and `HourHand` and their `move()` methods, but `tick()` could be arbitrarily complex.

**Cons:**
- Less flexible
  - Forced to accept superclass properties and methods, may have to (partially) override concrete methods, but overriding may have unintended consequences.
  - Because you don’t know how hidden functionality in superclass will affect your code.
  - And superclass can change implementation and accidentally affect you (see appendix!)

Interfaces + Composition

**Solution 2** uses a combination of an interface and composition to delegate functionality to a `MinuteHand` and `HourHand`.

**GrandfatherClock** signs the contract (thus has to implement `tick()` functionality) but delegates most of the responsibility to `MinuteHand` and `HourHand`.

```java
public interface Clock {
    void tick();
}

public class GrandfatherClock implements Clock {
    private HourHand hourHand;
    private MinuteHand minuteHand;

    public GrandfatherClock() {
        // instantiate HourHand and MinuteHand
    }

    @Override
    public void tick() {
        this.minuteHand.move();
        this.hourHand.move();
        if(this.isEvenHour()) {
            this.playDing();
        }
    }
}
```

Interfaces + Composition Design Pros

- **Very flexible**
  - We completely control `GrandfatherClock`, and if we want to write a `CuckooClock` or `DigitalClock` class, it’s easier to implement that functionality.
  - No overriding → no unintended consequences.

- **Easy to use classes written by others**
  - If someone else wrote `MinuteHand` and `HourHand`, you can still delegate to it without knowing their code details.
  - Could also easily swap them out with different component classes that you wrote.
**Interfaces + Composition Design Cons**

- Cons:
  - both inheritance and interface use composition (i.e., delegate to other objects)
  - with inheritance you automatically get concrete methods from the superclass
  - when you use composition, you must invoke the methods you want on the objects to which you have delegated — thus more control but more responsibility

**Case 3: Multiple Interfaces**

- Have seen how interfaces provide us with more flexibility because no unintended consequences
- Interfaces offer us even more flexibility because can implement several interfaces
- why is this useful?

Imagine we're making a game with the following classes:

- **FlyingSuperhero**
  - fly()
  - saveLives()

- **StrongSuperhero**
  - liftCars()
  - saveLives()

- **SlimeMonster**
  - scareCitizens()
  - oozeSlime()

- **Robber**
  - scareCitizens()
  - robBank()

**Interfaces vs. Inheritance**

- There are some similarities in implementation
  - **FlyingSuperhero** and **StrongSuperhero** both have a `saveLives()` method
  - **SlimeMonster** and **Robber** both have a `scareCitizens()` method
  - can abstract this up into superclasses!
Initial Design

![Initial Design Diagram]

Extending Our Design

- We want to add a monster who flies
  - FlyingMonster
    - fly()
    - scareCitizens()
- Where do we fit this into our inheritance diagrams?
  - It can fly, but it does not save lives — can’t use methods defined in Hero superclass to scareCitizens()
  - Could extend Villain superclass so that it can use scareCitizens(), but would need to reimplement code for fly()

Revised Design

![Revised Design Diagram]
Can we do better?

- Separate classes by their capabilities
  - FlyingSuperhero: flier + lifesaver
  - StrongSuperhero: carlifter + lifesaver
  - SlimeMonster: slimer + scarer
  - FlyingMonster: flier + scarer
  - BankRobber: robber + scarer
- Inheritance: model classes based on what they are
- Interface: model classes based on what they do
  - in this case, prefer interface over force-fitting inheritance

Better Design: Mix and Match Using Interfaces

Interfaces and Our Design

- As you can see, there are a lot more classes in this design
  - however, we have extreme flexibility
  - could make a flying, strong, scary, bank robbing monster
    without changing or force-fitting our new class into the current design
  - although you still have to implement the methods of the interface in your new class
The Challenges of Design (1/2)

- Design a solution to a problem such that it solves the problem efficiently, but also makes it easy to extend the solution if additional functionality is required
  - only define the capabilities that you know you will need to solve the problem at hand
- Your job in creating an interface/superclass is precisely to figure out the right abstractions
  - decision making under uncertainty – you do the best you can. And frankly, opinions may differ on what is "the best solution"
  - experience (practice) really matters
- Extensibility is important, but only to a degree
  - you cannot design a program that solves every problem a user thinks of

The Challenges of Design (2/2)

- CS32 (Software Engineering) goes deeper into design decisions and tradeoffs, as well as software engineering tools
  - you can take it after you’ve completed CS0150 and CS0200!

Overview

- Enums
- Factory Pattern
- Testing
- Interfaces vs. Inheritance
- Method Overloading
Method Overloading (1/3)

- Can define multiple methods of same name within a class, as long as method signatures are different.
- Method signature refers to name, number, types of parameters and their order.
- Signature does NOT include return type.
- Two methods with identical signatures but different return types (and different bodies) will yield a compiler error – why?
  - Compiler (and you, the reader) can’t distinguish between two methods with the same signature and different return types when an instance calls those methods – method name and argument types passed in are the same! So, signature is just name and parameter list.

TopHat Question

Which of the following is true of a class that contains an overloaded method? The class has...

A. Two methods that are absolutely identical.
B. Two methods that are the same, except in their return type.
C. Two methods that have the same name, but different parameters.
D. Two methods that are the same, except one contains an error.

Method Overloading (2/3)

- Example: java.lang.Math
  - Static method max takes in two numbers and returns the greater of the two.
  - There are actually three max methods – one for ints, one for floats, one for doubles.
  - When you call an overloaded method, the compiler infers which method you mean based on types and number of arguments provided.

```java
/* This is an approximation of what Math’s three max method look like */
public class Math {
    // other code elided
    public static int max(int a, int b) {
        // return max of two ints
    }
    public static float max(float a, float b) {
        // return max of two floats
    }
    public static double max(double a, double b){
        // return max of two doubles
    }
}
```
Method Overloading (3/3)

- Be careful not to confuse overloading and overriding!
  - Overriding an inherited method in a subclass: signatures and return types must be the same
  - Overloading methods within the same class: names are the same but the rest of the signatures (i.e., the parameters) must be different so the compiler can differentiate; the return types may also differ (see max)
  - Using same signatures and return types in different classes is OK because the compiler can differentiate by class/type of instance on which the method is called

Method Overloading: Constructors

- Even constructors can be overloaded!
- Already seen this with JavaFX shapes
- Can instantiate a rectangle with any of the constructors!

```java
Rectangle rect = new Rectangle();
rect = new Rectangle(120, 360);
rect = new Rectangle(0, 0, 120, 120);
rect = new Rectangle(Color.BLUE);
```

Method Overloading: Example

- Can call an overloaded method on other overloaded methods

```java
public class Halloween {
    public Halloween(HalloweenShop shop) {
        Hat hat = shop.getHat();
        this.wearCostume(hat);
    }
    public void wearCostume(Hat hat) {
        Gown gown = hat.getMatchingGown();
        this.wearCostume(hat, gown);
    }
    public void wearCostume(Hat hat, Gown gown) {
        //code to wearCostume elided
    }
    //other methods elided
}
```
Announcements

- Snake Code on GitHub – can discuss design decisions with other students, or TAs at hours
- DoodleJump Information
  - Early handin: Monday 10/30
  - On-time handin: Wednesday 11/01
  - Late handin: Friday 11/03
  - Check out Partner Projects Logistics Guide
  - Chance for a Code Debrief after you hand in the project! Will send more info soon!

- Debugging code-along—Check Ed!
  - Most important of the year!! Will save you hours on Tetris/Final Project
- HTA office hours on Friday 10/27 @3pm in CIT 210

Socially Responsible Computing

Social Media & Misinformation

CS15 Fall 2023

How do you receive your news?

Source: Pew Research Center
Misinformation / Disinformation

• July 2021: Biden: Facebook is ‘killing people’ because of vaccine hesitancy
  
  How much is FB to blame?

• July 2023: Louisiana judge rules that gov. agencies cannot communicate w/ social media platforms about deletion of posts

  Source: NYTimes

Why is this content popular?

• Social network algorithms tend to reward extreme content!
  
  • Shock → more engagement → more revenue
  
  • Confirmation bias
  
  • Filter bubble: when an internet user encounters only information that reinforce their own beliefs
  
  • AKA “echo chamber”
  
  • Contrary evidence can harden a belief: “post-truth” world of alternative facts

  Result: tribalism, divisiveness, polarization in the US, decline of civic responsibility
Social Media Can Reward Sharing Fake News

- In a study of 2,476 Facebook users, 30%+ of the false news shared was due to the 15% most habitual news sharers.
- Social media has a rewards system (likes, etc.) that encourages users to keep posting attention-grabbing content — like a video game.

X (formerly Twitter) Discussion

- Jan 2021: Trump permanently banned from Twitter & other platforms
- Nov 2022: Elon Musk ends Trump's ban after posting a poll asking if Trump should be allowed back.

Source: NBC News

Moderating the Spread of Terrible News

Graphic war videos go viral, testing social media's rules.

Israel-Hamas war: Tech platforms under scrutiny over spread of false, graphic posts.

Sources: Washington Post, ABC
We can benefit from algorithmic detection and throttling + human content moderation…

but ultimately, harmful content isn’t only a technical problem; it stems from the social problem of actions that want to spread mis- and dis-information.
• Assume Car uses its method revEngine() (which uses Engine’s rev()) inside its definition of drive

```java
public class Car {
    private Engine engine;
    private Brakes brakes;
    public Car() {
        this.brakes = new Brakes();
        this.engine = new Engine();
    }
    public void revEngine() {
        this.brakes.engage();
        this.engine.rev();
    }
    public void drive() {
        this.revEngine();
        this.brakes.disengage();
        //remaining code elided
    }
}
```

```java
public class Brakes {
    //constructor, other code elided
    public void engage() {
        //code elided
    }
    public void disengage() {
        //code elided
    }
}
```

```java
public class Engine {
    //constructor, other code elided
    public void rev() {
        //code elided
    }
}
```

• Now we override revEngine in ElectricCar
  • notice revEngine no longer calls brakes.engage()
  • Recall that drive() calls revEngine; if you call drive() on ElectricCar, it will call Car’s inherited drive() that uses ElectricCar’s revEngine implementation

```java
public class ElectricCar extends Car {
    private Battery battery;
    public ElectricCar() {
        super();
        this.battery = new Battery();
    }
    @Override
    public void revEngine() {
        this.battery.usePower();
    }
}
```

• This could pose a problem
  o drive() relies on revEngine to engage the brakes, so that drive() can disengage them, but you don’t know that – hidden code
  o so when ElectricCar overrides revEngine(), it messes up drive()
  o ElectricCar also has 2 engines now
    o its own Battery and the pseudo-inherited engine from Car
    o also messes up its functionality
  • It might be fine if you write all your own code and know exactly how everything works
  o but usually not the case!

```java
public class Car {
    //code elided
    public void revEngine() {
        this.brakes.engage();
        this.engine.rev();
    }
    public void drive() {
        this.revEngine();
        this.brakes.disengage();
        //remaining code elided
    }
}
```

```java
public class ElectricCar extends Car {
    private Battery battery;
    public ElectricCar() {
        super();
        this.battery = new Battery();
    }
    @Override
    public void revEngine() {
        this.battery.usePower();
    }
}
```

```java
public class Brakes {
    //constructor, other code elided
    public void engage() {
        //code elided
    }
    public void disengage() {
        //code elided
    }
}
```

```java
public class Engine {
    //constructor, other code elided
    public void rev() {
        //code elided
    }
}
```