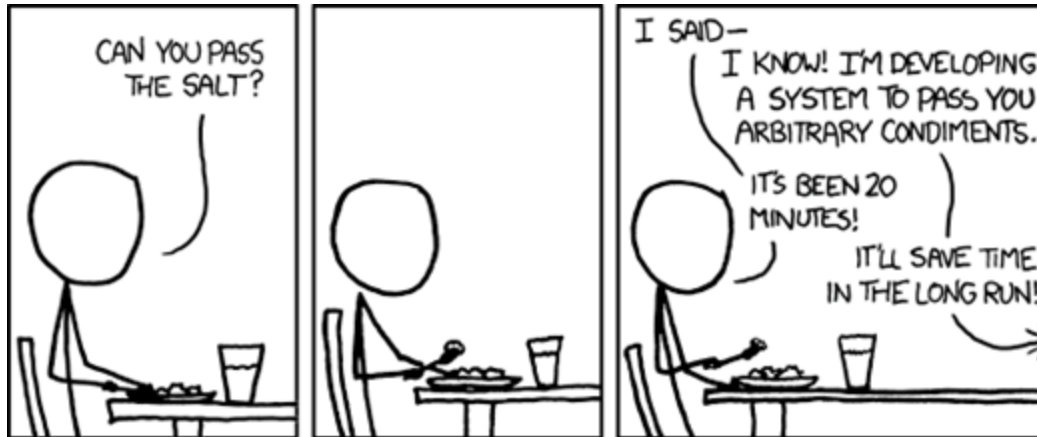


# Lecture 15

## Design Patterns and Principles: Part 2



From xkcd Webcomics

# Use of ChatGPT and other non-ATA GAI

- Strictly forbidden in CS15. See: collaboration policy
- If ATA doesn't satisfy you, use Ed or go to conceptual/debugging hours
- We can easily detect use of ChatGPT, etc., in your code



# 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 180°
  - 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

# 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
- Pro: easy to read and understand
- Con: 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

```
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
- Pro: it is less likely to mistype an integer compared to a string
- Con: 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

```
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



# Declaring and Defining an Enum

- **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 icon E
- 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 currDirection = Direction.UP;
```

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



# 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 `Direction.RIGHT` in constructor
  - even though we only use `currDirection` in the `changeDirection` method, still instance variable because it is an attribute of the class
- 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 `ints`!

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

# TopHat Question

Join Code: 316062

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

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

- A. `Status gameStatus = new Status(PAUSED);`
- B. `Status gameStatus = Status(PAUSED);`
- C. `Status gameStatus = Status.PAUSED();`
- D. `Status gameStatus = 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 `KeyCodes` 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 180°
  - thus check new direction passed from key input is not the opposite of current direction

```
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;  
        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, messy 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(Direction current) {  
        switch (current) {  
            case UP:  
                return DOWN;  
            case DOWN:  
                return UP;  
            case LEFT:  
                return RIGHT;  
            case RIGHT:  
                return LEFT;  
        }  
    }  
}
```

*note: we don't need a `break` at the end of each case of the switch statement because we are returning from the method!*

# Enum Methods (2/3)

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

*this* is the current value of **currDirection**. When **opposite()** is called, we check said current direction and return its opposite

```
public enum Direction {  
    UP, DOWN, LEFT, RIGHT;  
  
    public Direction opposite() {  
        switch (this) {  
            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

```
public class Snake {  
  
    private Direction currDirection;  
  
    //initialize currDirection to RIGHT  
    public Snake() {  
        this.currDirection = Direction.RIGHT;  
    }  
  
    public void changeDirection(Direction newDir) {  
        if (newDir != this.currDirection.opposite()) {  
            this.currDirection = newDir;  
        }  
    }  
  
}
```



# TopHat Question

Join Code: 316062

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

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

A.

```
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;  
    }  
}
```

B.

```
public int calc(int a, int b) {  
    switch(this) {  
        case 1:  
            return a + b;  
        case 2:  
            return a - b;  
        case 3:  
            return a * b;  
        case 4:  
            return a / b;  
    }  
}
```

C.

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

# Overview

- Enums
- Factory Pattern
- Testing
- Interfaces v. 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



## Pellet Constructor!

```
public Pellet(Pane gamePane, Color color, int score,  
int row, int col)
```

```
public void spawnFood() {  
    // gets random empty tile on board where food will be added  
    BoardSquare tile = this.getRandomEmptyTile();  
    Pellet 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

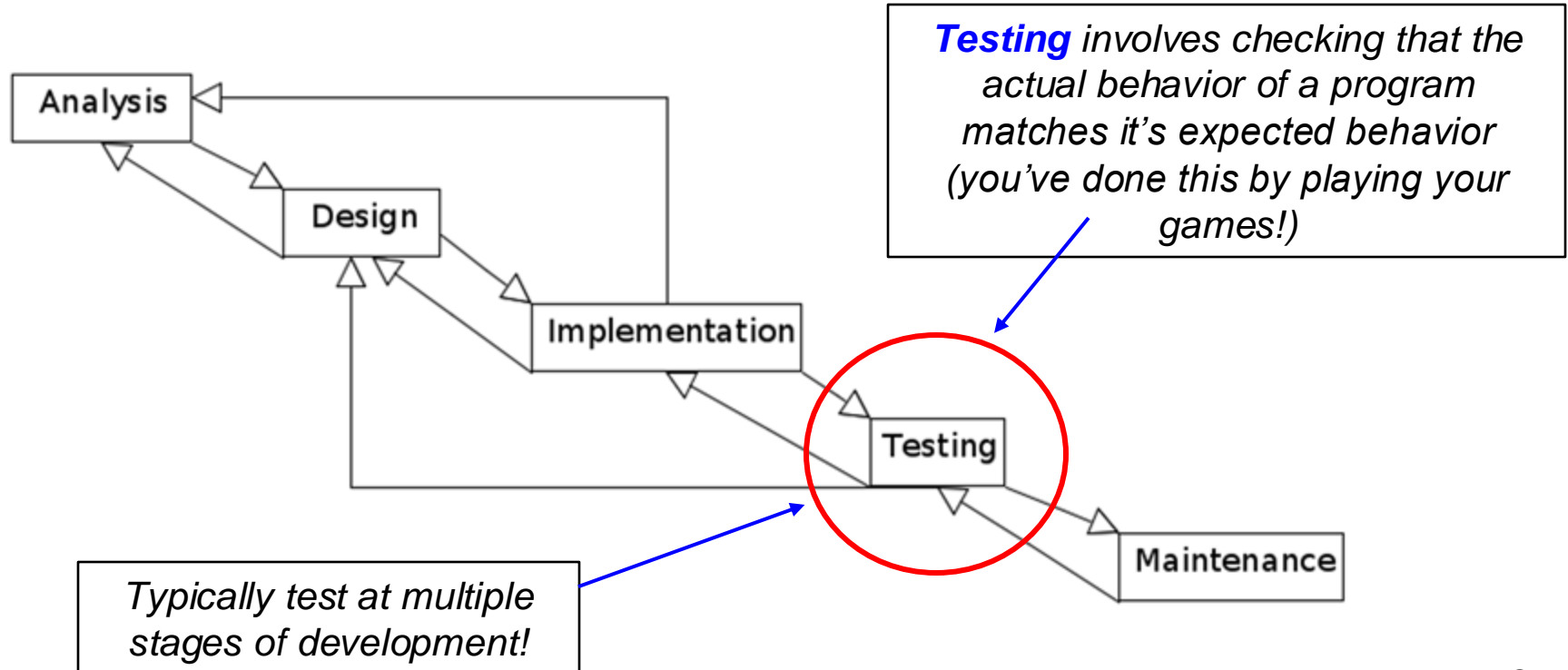
- Enums
- Factory Pattern
- Testing
- Interfaces v. Inheritance
- Method Overloading





# Testing Our Program (1/2)

## Original “Waterfall Model” of Software Development



# Testing Our Program (2/2)

- You already test your programs all the time – by playing them!
- As we scale in complexity, we can't wait until we have all of the pieces put together to test individual components
  - we want to be incrementally developing, testing components, i.e., "units" one at a time
- **Unit Testing** is useful for verifying that specific parts of our program work (ex. a method)
- 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 `println`s, 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 one 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 set up and practice with JUnit in this week's lab (Check out Appendix A for more details!)

# JUnit Testing: Snake Example

## How can we apply the JUnit Testing 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 this and next week's labs
  - You will learn a lot more about unit testing in CS200!! 😊

```
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());  
  
    }  
}
```

# Recap Snake Design Process

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

# Intermission

- Have seen how to design mock CS15 project from scratch
  - need to go through similar design discussions for the projects in the remainder of the semester
  - 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 v. Inheritance
- Method Overloading



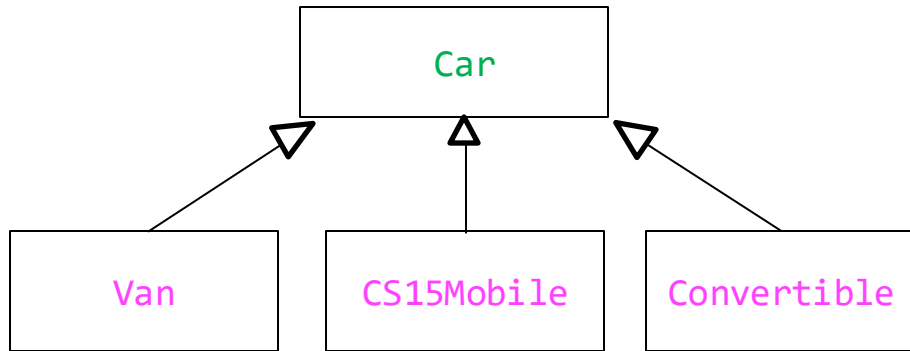
# Interfaces vs. Inheritance

- When deciding between interfaces and inheritance, need to consider trade-offs between the two
  - interfaces + composition 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 interfaces (coupled with composition) are 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 engines
- 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** doesn't 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 B)
    - when you create a subclass of **Car**, its component **this.engine**, is hidden from you
      - a parent's private variables stay private
    - you inherit methods that use **this.engine**, but their 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

```
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

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

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

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

```
public interface Clock {
    public 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();
        }
    }
}
```

# 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 `tick()`. In this case, `tick()` delegates 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
  - you don't know how hidden functionality in superclass will affect your code
  - superclass can change implementation and accidentally affect you (see Appendix B!)

# 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`

```
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()

SlimeMonster

- scareCitizens()
- oozeSlime()

StrongSuperhero

- liftCars()
- saveLives()

Robber

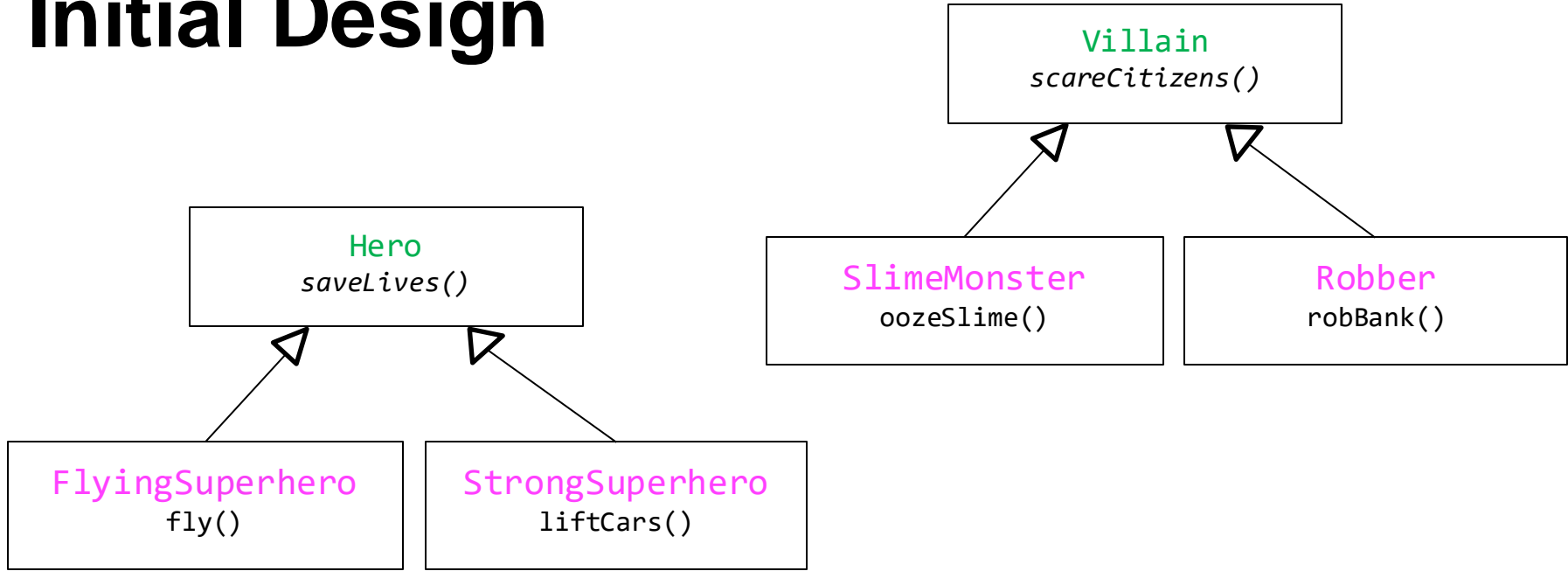
- scareCitizens()
- robBank()

# Interfaces vs. Inheritance

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



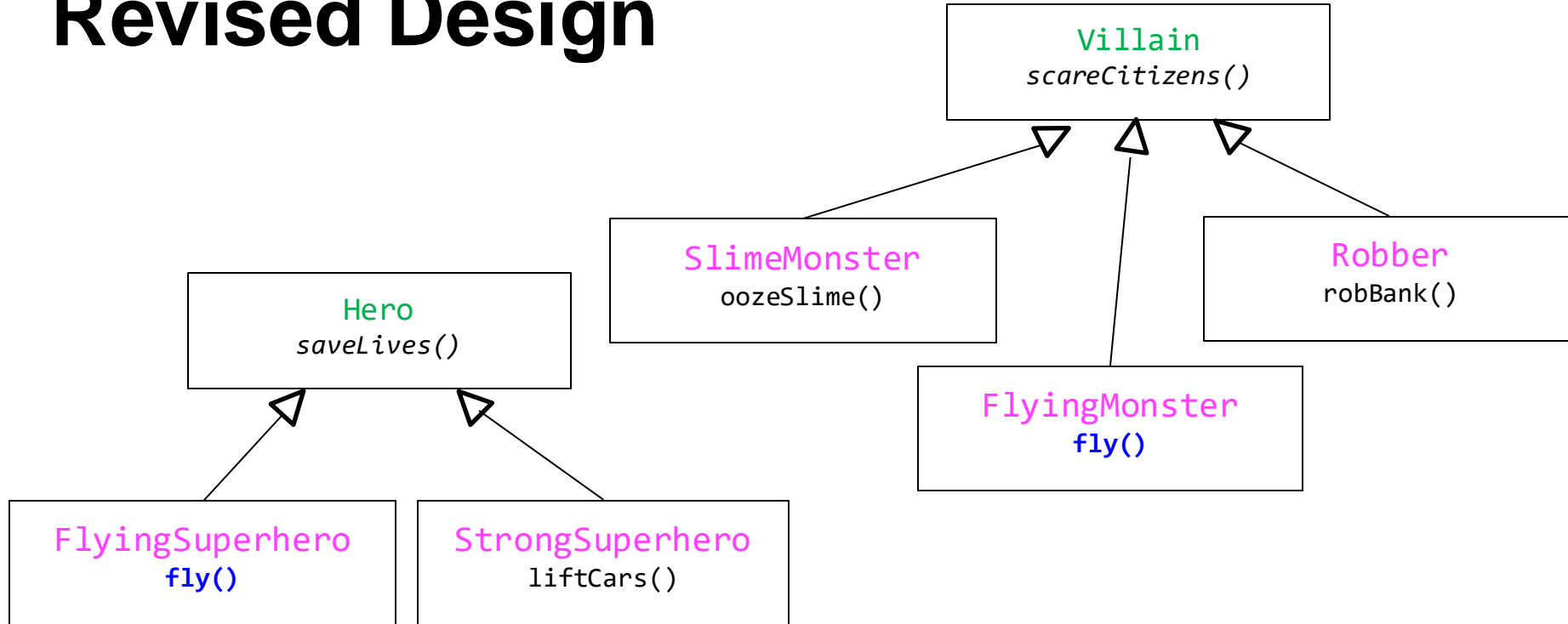
# Initial Design



# 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()`

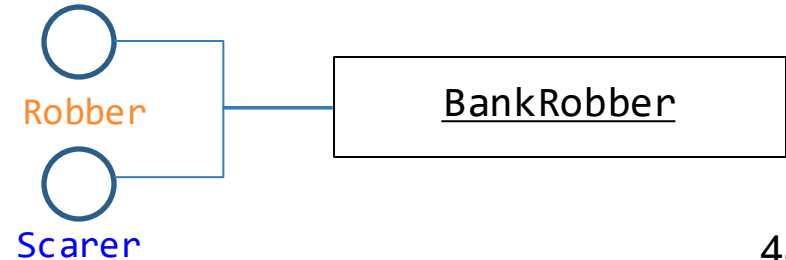
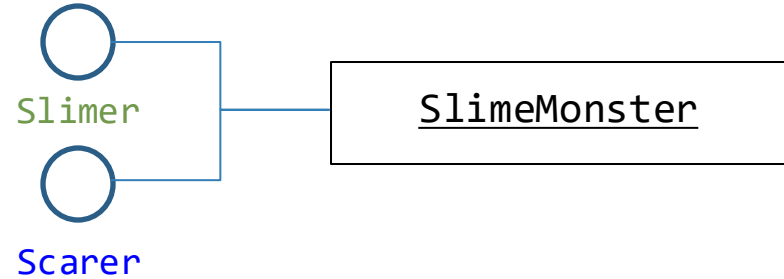
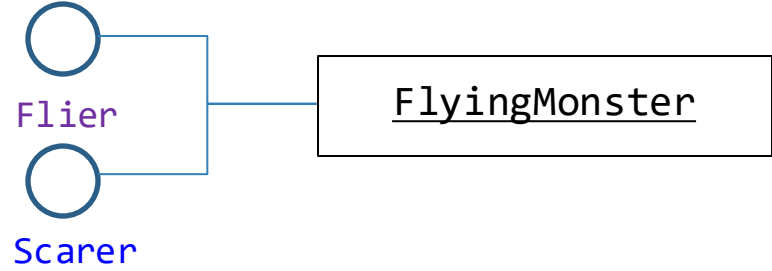
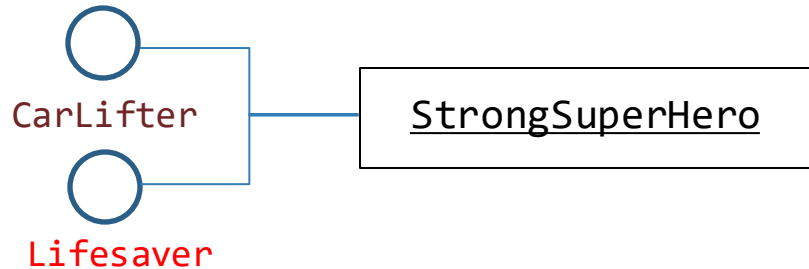
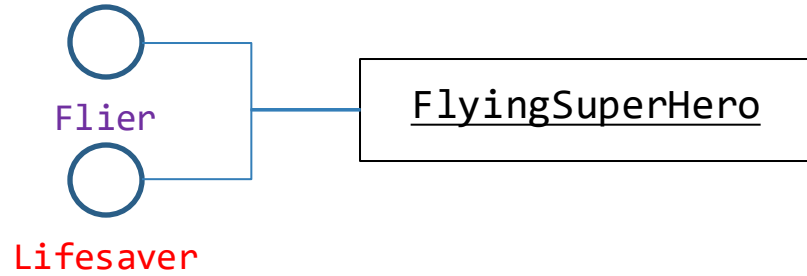
# Revised Design



# 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 v. 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**

# 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

```
/* this is an approximation of what Math's  
three max methods 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

# TopHat Question

**Join Code: 316062**

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: Constructors

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

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

## Constructor Summary

### Constructors

#### Constructor and Description

##### **Rectangle()**

Creates an empty instance of Rectangle.

##### **Rectangle(double width, double height)**

Creates a new instance of Rectangle with the given size.

##### **Rectangle(double x, double y, double width, double height)**

Creates a new instance of Rectangle with the given position and size.

##### **Rectangle(double width, double height, **Paint** fill)**

Creates a new instance of Rectangle with the given size and fill.

# Method Overloading: Example

- Can call an overloaded method on other overloaded methods

```
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 10/28
  - On-time handin: Wednesday 10/30
  - Late handin: Friday 11/01
  - Check out Partner Projects Logistics Guide
  - Chance for a Code Debrief after you hand in the project! Will send more info soon!
- HTA office hours on Friday 10/25 @3pm in CIT 209

# Appendix A: JUnit Testing!

# JUnit Testing: Naïve Example

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

*@Test tells compiler this is a unit test*



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

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

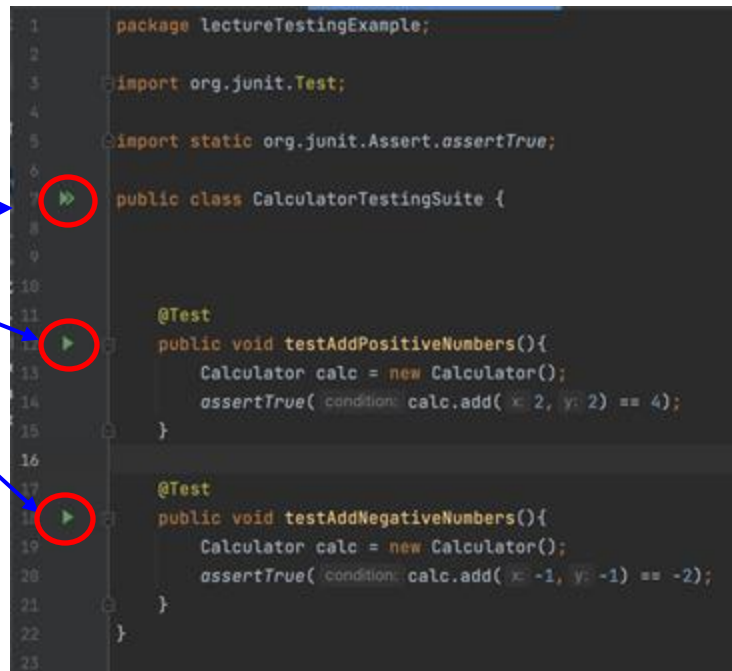
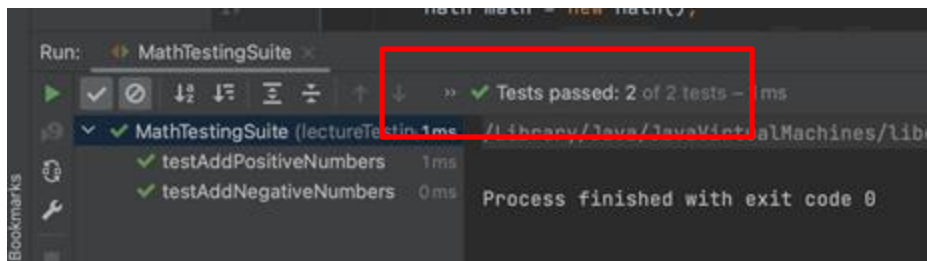
```
public class CalculatorTestingSuite {  
  
    @Test  
    public void testAddNumbers(){  
        Calculator calc = new Calculator();  
        assertTrue(calc.add(2,2) == 4);  
    }  
}
```

# How does IntelliJ help?

- Our test(s) 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:**



# Appendix B: Method Overriding!

# Unintended Consequences of Overriding (1/3)

- Assume `Car` uses its method `revEngine()` (which uses `Engine`'s `rev()`) inside its definition of `drive`



```
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
    }
}
```

```
public class Brakes {
    //constructor, other code elided

    public void engage() {
        //code elided
    }

    public void disengage() {
        //code elided
    }
}

public class Engine {
    //constructor, other code elided

    public void rev() {
        //code elided
    }
}
```



# Unintended Consequences of Overriding (2/3)

- 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

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

---

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

# Unintended Consequences of Overriding (3/3)

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

```
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  
    }  
}
```

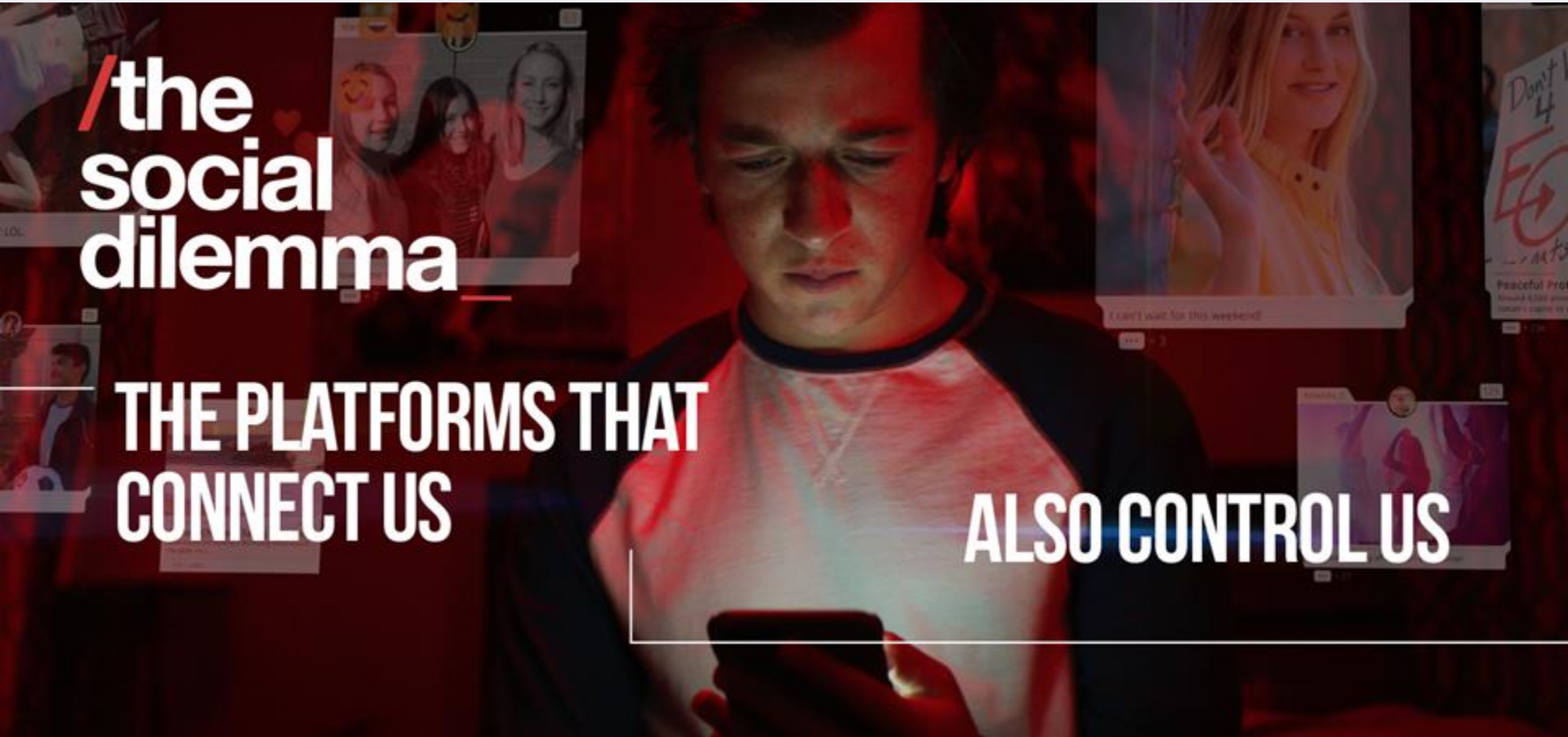
```
public class ElectricCar extends Car {  
    private Battery battery;  
    public ElectricCar () {  
        this.battery = new Battery();  
    }  
    @Override  
    public void revEngine() {  
        this.battery.usePower();  
    }  
}
```

# Social Media I: Misinformation

Topics in Socially Responsible Computing



# The Social Dilemma – SRC Discussion #1

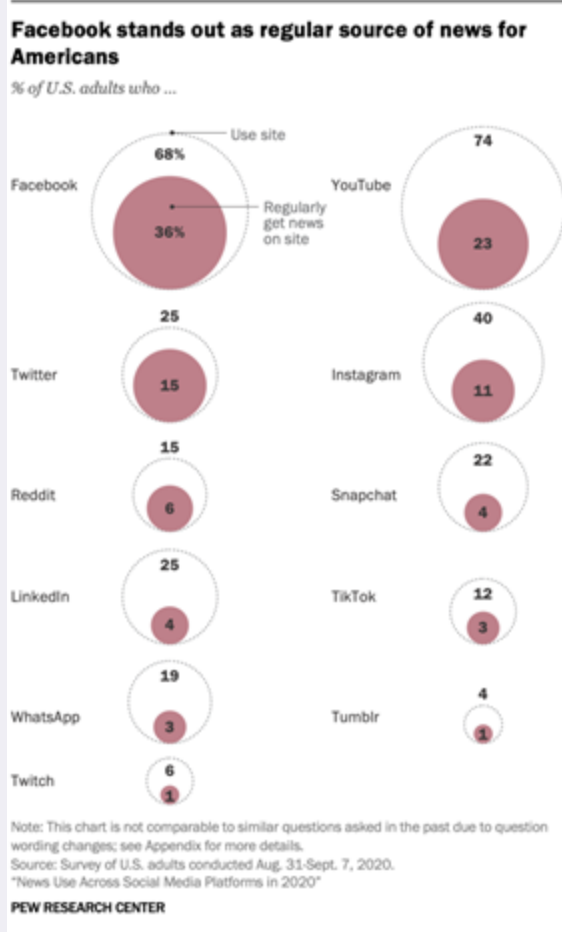


**/the  
social  
dilemma**

**THE PLATFORMS THAT  
CONNECT US**

**ALSO CONTROL US**

# Where do you receive your news?



WhatsApp



Vassar

Jun 1, 2020

So he said it, wor  
it's a joke. 🖊 It's  
that chance to al  
they have ever d  
humanity in any  
See more

# BILL GATE THE COVID USE EXPERIMENTAL TECHNOLOGY AND PERM DNA

May 15, 2020



John Sipher Retweeted

**JohnCammo** @JohnCa

Replying to @joshtpm

Trump's October surpr  
Fake, but quite dramati  
the "Trump COVID wat  
Then, 14 days later, Tru  
hydroxychloroquine.

338 5.7K



**End of Days Radio** @ninjashoes · Jun 7

5G Corona is the #truth There is no damn viru  
same time they rolled out 5G. The other factor  
dust they use to strengthen the signal and bo

Get the facts about COVID-19



**Tony "Abolish (Pol)ICE" Arcieri** @bascul

Replying to @hdevalence

WHO is misinformation, 5g corona is the truth

Tweet

**Keri Hilson** @KeriHilson

People have been trying to warn us  
about 5G for YEARS. Petitions,  
organizations, studies...what we're  
going thru is the affects of radiation.  
5G launched in CHINA. Nov 1, 2019.  
People dropped dead. See attached &  
go to my IG stories for more. TURN  
OFF 5G by disabling LTE!!!

Who invented 5G?

China

China has launched its 5G national network and started commercial operation on 1 November 2019. At launch, Chinese state media called it the world's largest 5G network.

China: Three Companies Lead the Charge

The world's largest 5G network was launched by the three largest Chinese network operators last 17, 2019, according to the open access operator Statista. China Mobile, China Unicom, and China Telecom all activated 5G networks in less than five months after they were issued 5G licenses. Each of the network operators offered their 5G services at 12 per cent less to 5G Chinese clients in the time of launching.

What we are seeing is a concerted effort by the Chinese to the economic situation and communication structure to the degree 5G is

Why is 5G dangerous?

Some 5G pundits contend that the new network generates radiofrequency radiation that can damage DNA and lead to cancer, cause oxidative damage that can cause premature aging, disrupt cell metabolism, and potentially lead to other diseases through the generation of stress proteins. Dec 16, 2019

Intertwined subsets of misinformation blend subliminally with conspiracy theories with a downplay of the seriousness of the pathogen causing the outbreak. In a 2019 Netflix documentary, Bill Gates warned against a global pandemic that could potentially originate in

8:58 PM · 3/15/20 · Twitter for iPhone

2,777 Retweets 5,557 Likes

'his infection."  
he screens,  
every day.

nd against tyranny and oppression.

ate  
y members can see who's in the group  
what they post.

ible  
one can find this group.

neral Group

planned.. several

eril are planned. More for the elderly. The  
plan is to kill all of us. Quickly. And  
yet... people are begging for it. Waiting  
for it. Waiting to hold up their precious  
children for it. Satan's biggest triumph.

8:29 AM · 7/24/20 · Twitter for iPhone

57 Retweets and comments 157 Likes

**john neck** @johncu64883844 · 13m

Replying to @KateShemirani

Some of my friends are thinking if the get  
the vaccine everything will go back to  
normal. Told them about deep state,  
Clinton's, pedowood and what's in the  
vaccine and there saying I don't care  
some think gates is amazing



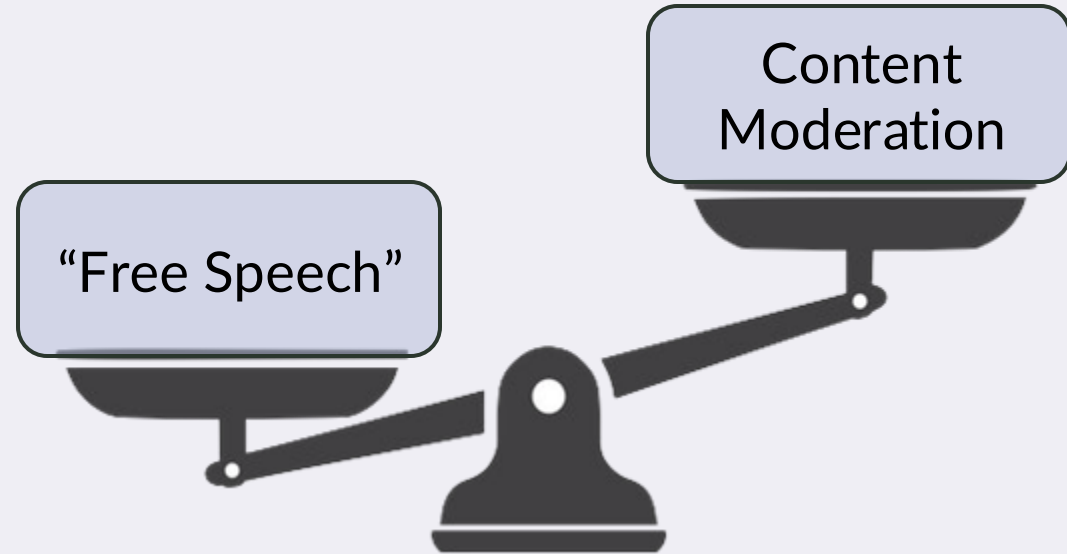
# Social Media, Misinformation, and Echo Chambers...

- **Algorithm Personalization:**
  - Similar content based on past behavior
- **Selective Exposure:**
  - Following accounts aligned with existing beliefs
- **Feedback Loop:**
  - More engagement, more similar content
- **Filter Bubbles:**
  - Limited exposure to opposing viewpoints




# The Digital Town Square...?

- Wide spectrum of opinions, but the trend has been to allow more “free speech”
  - Does this even lead to a so-called free “marketplace of ideas”?
- Balance between meaningful political discourse and regulating hate speech or misinformation
- Platforms like X preach *hyper-libertarianism* but ban accounts in a non-transparent and often-biased way



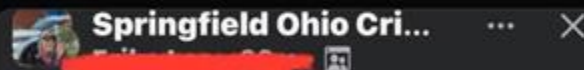


 LemmiesLuLu @BuckeyeGrrl · Sep 5  
Remember when my hometown of Springfield Ohio was all over National news for the Haitians?

I said all the ducks were disappearing from our parks?

Well, now it's your pets.

@FoxNews



Warning to all about our beloved pets & those around us!!

My neighbor informed me that her daughters friend had lost her cat. She checked pages, kennels, asked around, etc. One day she came home from work, as soon as she stepped out of her car, looked towards a neighbors house, where Haitians live, & saw her cat hanging from a branch, like you'd do a deer for butchering, & they were carving it up to eat. I've been told they are doing this to dogs, they have been doing it at snyder park with the ducks & geese, as I was told that last bit by Rangers & police. Please keep a close eye on these animals


 End Wokeness @EndWokeness

Springfield is a small town in Ohio.

4 years ago, they had 60k residents.

Under Harris and Biden, 20,000 Haitian immigrants were shipped to the town.

Now ducks and pets are disappearing.

 Springfield Ohio Cri...

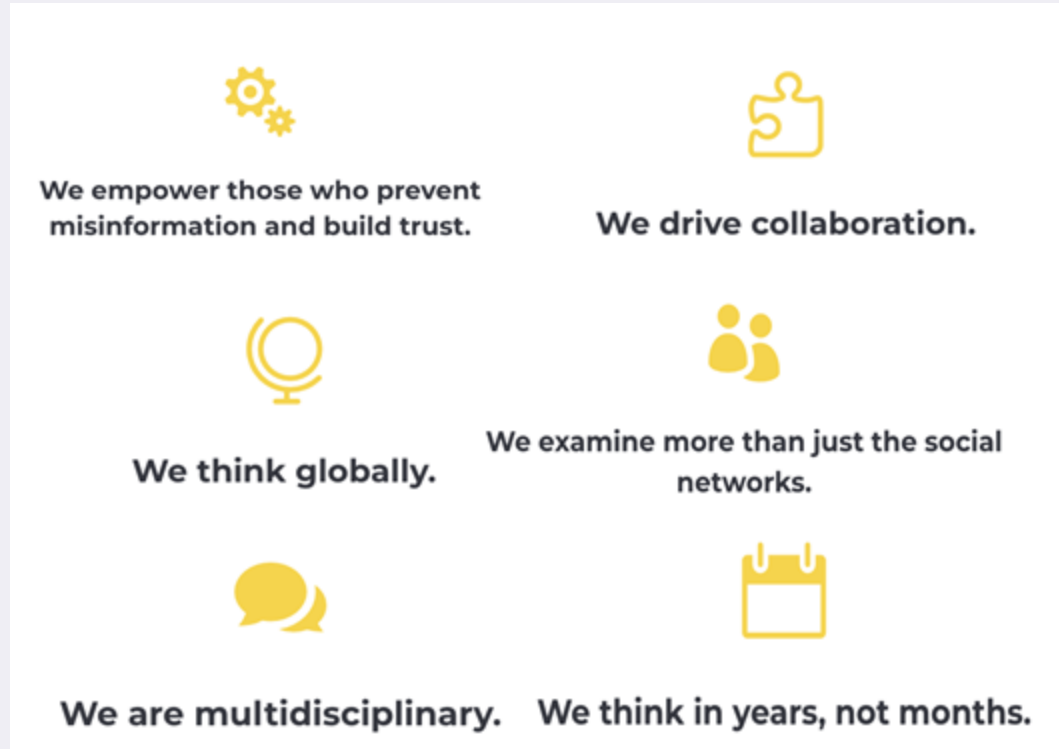
Warning to all about our bel & those around us!!  
My neighbor informed me th daughters friend had lost he checked pages, kennels, as around, etc. One day she ca from work, as soon as she s of her car, looked towards a house, where Haitians live, cat hanging from a branch, do a deer for butchering, & carving it up to eat. I've bee are doing this to dogs, they doing it at snyder park with & geese, as I was told that l Rangers & police. Please ke eye on these animals

Trump falsely accuses immigrants in Ohio of abducting and eating pets



8:13 PM · Sep 6, 2024 · 4.9M Views

# Information Futures Lab @Brown



<https://sites.brown.edu/informationfutures/>