Lecture 15

Design Patterns and Principles: Part 2

From xkcd Webcomics: https://xkcd.com/974/
Overview

● ** Enums **
● ** Factory Pattern **
● ** Interfaces v. Inheritance **
● ** Method Overloading **
Back to Our Snake Program

• Specifications
  o player moves snake via key input around board of squares with goal of eating pellets to increase score
  o snake can pivot left or right but not 180º
  o 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 also delegate to a wrapper Board class
• Today, we’ll cover details about snake movement and food
Overview

- Enums
- Factory Pattern
- 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.
- The direction in which the snake is moving is its *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!
  o e.g., the north direction can be represented as “up”, “upward”, “UP”, “upside” and many more
  o 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
• Pro: it is less likely to mistype an integer compared to a string
• Con: the numbers used are arbitrary
  o e.g., 1 can mean anything. If 1 is up, is down -1 or 2?
  o 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**
  o 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
  o e.g., the cardinal directions: north, south, east, west
  o can create a **Direction** enum for this (we’ll see how to do this next)
• The value of an enum can be any of the constants pre-defined for it
  o the value of the **Direction** enum would be any of 4 directions
• In our program, we can use enums to represent the direction of snake movement
  o snake moves up, down, left or right. Since these values are constant, we can represent them using enums
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 UPPERCASE letters
- To access the enum constants, use the dot syntax:
  ```java
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 icon E

```java
public enum Direction {
    UP, DOWN, LEFT, RIGHT;
}
```
Using Enums: Snake Movement (1/3)

• Can use `Direction` enum in `Snake` to store direction of movement
  o notice `currDirection`’s type is the `enum Direction`. Not `String` or `int`
  o `currDirection` is initialized to right

• Like any type in Java, enums can be used as parameters to methods
  o `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!

• Enums are a great way of representing a set of related constants used in a program

```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)

• To handle key input, use a switch statement and call `changeDirection` in each case, passing in the matching direction
  
  o `KeyCode` is actually an `enum` defined by Java FX whose constants are the different keys e.g., UP, DOWN, ... for the arrow keys, ENTER, SPACE, etc.

• But wait! There’s one specification with snake movement we’ve ignored
  
  o snake can pivot right or left, but not 180°

• Need to check that the new direction passed from key input is not the opposite of current direction

```java
private void handleKeyInput(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.LEFT);
            break;
        default:
            break;
    }
}
```
• Can use a series of if-else statements to check that newDir is not the direction opposite currDirection

• Results in complicated code which would be hard to debug if one of the directions was mistyped

• We need a simpler solution
  o given a direction, can we find its opposite?
  o how can we have this functionality be part of the enum so that snake can use it?

```java
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
  o not a class, no constructor because values already enumerated at the beginning
• Can add a method, **opposite**, in our **enum**, that returns the opposite direction of the current direction
• But need to know what current direction is
  o can pass it to **opposite** as a parameter. Anything wrong with this?
  o repetitive since would call: `currDirection.opposite(currDirection);`

```java
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;
        }
    }
}
```
Enum Methods (2/3)

• Can instead pass `this` to switch statement
  o i.e., the value of the `enum` we’re calling `opposite` on:
    `current.opposite();`
  o analogous to using `this` inside a class to refer to itself

• If `current` is `Direction.LEFT`, what would `current.opposite()` return?

```java
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;
        }
    }

    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 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

```java
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

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

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

public int calculate(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
● 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
  - if there were different actions associated with each **pellet**, we might want to use an **interface**
Representing the Pellets (3/3)

• Can use **inheritance** and factor out common implementation to superclass 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 bringing 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 (coming up in a few slides!)

• Any other option?
  - recall how we generated different types of **Bandanas** 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!)
    - don’t need to know how the different types are created. Let **Pellet** class deal with that
Factory Pattern (2/2)

• **Key features**: often a random number generator typically a switch statement that creates cases in which one of many related classes are instantiated

```java
public void spawnFood() {
    // gets random empty tile on board where the 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,
                               tile.getRow(), tile.getCol());
            break;
        case 1:
            food = new Pellet(this.gamePane,
                               Color.BLUE,
                               tile.getRow(), tile.getCol());
            break;
        default:
            food = new Pellet(this.gamePane,
                               Color.GREEN,
                               tile.getRow(), tile.getCol());
            break;
    }
    tile.addFood(food);
}
```
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?
  - is my design scalable?
  - 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
- Interfaces v. Inheritance
- Method Overloading
Interfaces vs. Inheritance

• When deciding between interfaces and inheritance, need to consider trade-offs between the two
  o while inheritance allows sub-classes to inherit functionality from parent, there’s risk of unintended consequences when overriding methods
  o interfaces + composition offer more flexibility compared to inheritance
    o can implement several interfaces but only extend one super-class
• Will look at case studies for each of these trade-offs
• Note that while interface (coupled with composition) is often favored over inheritance, there are use cases which can really take advantage of inheritance
Case 1: Risks 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
  o start/stop engines
• We created a Car superclass
  o Car contains instances of Engine, Brake, etc.
  o CS15Mobile, Van, and Convertible extend from Car
Extending Our Design

• Assume now that we add an ElectricCar class to the program
  o but ElectricCar doesn’t use the standard Engine inherited from Car
  o can ElectricCar just override Car’s methods that make use of Engine? Anything wrong with that?
    ▪ can do this but could be dangerous
  o when you subclass Car, the instance of Engine, engine, is hidden from you
    ▪ a parent’s private variables stay private
  o you inherit these methods that use engine, but their implementation is hidden from you
    ▪ you do not know which methods use engine, let alone how they do that
Unintended Consequences of Overriding (1/3)

- Assume `Car` uses its method `revEngine()` (which uses `Engine`'s `rev()`) inside its definition of `drive`
Unintended Consequences of Overriding (2/3)

• Now we override `revEngine` in `ElectricCar`
  o 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
Unintended Consequences of Overriding (3/3)

• 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
    ▪ 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
  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 () {
        this.battery = new Battery();
    }
    @Override
    public void revEngine() {
        this.battery.usePower();
    }
}
```
Case 2: Interfaces + Composition

- Seen how inheritance can break functionality in sub-classes. But how, if at all, are interfaces 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?

```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 */
    }
}

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
  o in solution 1, Clock is a superclass
  o in solution 2, Clock is an interface
  o both can be used polymorphically

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

Pros:
• Better code reuse
  o by extending superclass, methods are automatically inherited in subclasses, so no need to re-implement functionality (tick()). In this case, delegates most of the responsibility to a MinuteHand and HourHand and their move() methods, but tick() could be arbitrarily complex)

Cons:
• Less flexible
  o forced to accept superclass properties and methods, may have to (partially) override concrete methods, but overriding may have unintended consequences, as in the previous example
  o because you don’t know how hidden functionality in superclass will affect your code
  o and they can change implementation and accidentally effect you
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
  o we completely control GrandfatherClock, and if we want to write a CuckooClock or DigitalClock class, it’s easier to implement that functionality
  o no overriding → no unintended consequences

• Easy to use classes written by others
  o if someone else wrote MinuteHand and HourHand, you can still delegate to it without knowing their code details
  o could also easily swap them out with different component classes that you wrote
Interfaces + Composition Design Cons

• Cons
  o both inheritance and interface use composition (i.e., delegate to other objects)
    ▪ but with inheritance you automatically get concrete methods from the superclass
    ▪ whereas when you do explicit composition, you have to 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 we can implement several interfaces
  - why is this useful?
- Imagine we’re making a game with the following classes

<table>
<thead>
<tr>
<th>FlyingSuperhero</th>
<th>SlimeMonster</th>
</tr>
</thead>
<tbody>
<tr>
<td>fly()</td>
<td>scareCitizens()</td>
</tr>
<tr>
<td>o saveLives()</td>
<td>o oozeSlime()</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>StrongSuperhero</th>
<th>Robber</th>
</tr>
</thead>
<tbody>
<tr>
<td>liftCars()</td>
<td>scareCitizens()</td>
</tr>
<tr>
<td>o saveLives()</td>
<td>o robBank()</td>
</tr>
</tbody>
</table>
Interfaces vs. Inheritance

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

Hero
- `saveLives()`

FlyingSuperhero
- `fly()`

StrongSuperhero
- `liftCars()`

Villain
- `scareCitizens()`

SlimeMonster
- `oozeSlime()`

Robber
- `robBank()`
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 → won’t use methods defined in Hero superclass so that it can use scareCitizens()
  - could extend Villain superclass so that it can use scareCitizens(), but would need to reimplement code for fly()
Revised Design

Hero
  saveLives()

FlyingSuperhero
  fly()

StrongSuperhero
  liftCars()

SlimeMonster
  oozeSlime()

Robber
  robBank()

Villain
  scareCitizens()

FlyingMonster
  fly()
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

Flier
Lifesaver

FlyingSuperHero

CarLifter
Lifesaver

StrongSuperHero

Flier
Scarer

FlyingMonster

Slimer
Scarer

SlimeMonster

Robber
Scarer

BankRobber
Interfaces and Our Design

• As you can see, there are a lot more classes in this design
  o 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
  o 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
  o decision making under uncertainty – you have to do the best you can. And frankly, opinions may differ on what is “the best solution”
  o experience (practice) really matters
• Extensibility is important, but only to a degree
  o 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/CS0200!
Overview

- Enums
- Factory Pattern
- 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?
  
  o 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 exactly 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 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
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 (0, 0, Color.BLUE);
```

### Constructor Summary

<table>
<thead>
<tr>
<th>Constructor and Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Rectangle()</strong></td>
</tr>
<tr>
<td>Creates an empty instance of Rectangle.</td>
</tr>
<tr>
<td><strong>Rectangle(double width, double height)</strong></td>
</tr>
<tr>
<td>Creates a new instance of Rectangle with the given size.</td>
</tr>
<tr>
<td><strong>Rectangle(double x, double y, double width, double height)</strong></td>
</tr>
<tr>
<td>Creates a new instance of Rectangle with the given position and size.</td>
</tr>
<tr>
<td><strong>Rectangle(double width, double height, Paint fill)</strong></td>
</tr>
<tr>
<td>Creates a new instance of Rectangle with the given size and fill.</td>
</tr>
</tbody>
</table>
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
  o Early handin: Monday 11/1
  o On-time handin: Wednesday 11/3
  o Late handin: Friday 11/5
  o Check out **Partner Projects Logistics Guide**
    o Individual check-ins after you hand in the project! Will send more info soon!
• **Tetris Partner Matching Form** released today
  o Cannot have same partner twice
• HTA office hours on Friday 10/29 @3pm in Friedman 101
Topics in Socially Responsible Computing

Artificial Intelligence II
Further Use of AI — ReCAPTCHA

- reCAPTCHA used to protect websites from bots with tests only humans should be able to pass
  - However: also used to build Google’s machine learning data sets
- “reCAPTCHA also makes positive use of the human effort spent in solving CAPTCHAs by using the solutions to digitize text, annotate images, and build machine-learning datasets. This in turn helps preserve books, improve maps, and solve hard AI problems.” – Google reCAPTCHA Webpage
- Speculation as to what data is being used for
  - Improving maps, self-driving car
Potemkin AI and Mechanical Turk (1/3)

• Mechanical Turk: Hungarian inventor Wolfgang von Kempelen unveiled an “exotic” automaton that could play chess by its own accord
  • Inside was a human chess master who used magnets and levers to operate Mechanical Turk
  • Complex mechanical system designed to distract from human labor

• Potemkin AI: building and presenting “AI systems” that are actually powered by human beings
  • Potemkin village: an impressive façade designed to hide undesirable conditions
Potemkin AI and Mechanical Turk (2/3)

- Many “AI” systems actually rely on humans
  - Autonomous vehicles using remote-driving
  - Humans read private emails for email-based services like personalized ads, price comparisons
  - SpinVox accused of using humans not machines to transcribe audio
  - Facebook personal assistant M used humans (before being shut down)

- Disguise true state of things and deceive potential customers

- Appealing to potential venture capital investors → motivated by profit and power
  - Illusion of complete control
    - Ex. surveillance systems
Potemkin AI and Mechanical Turk (3/3)

• Amazon Mechanical Turk (MTurk)
  • Crowdsourcing marketplace that “makes it easier for individuals and businesses to outsource their processes and jobs to a distributed workforce” (mturk.com)

• Makes micro-gig work possible
  • Crowdsourcing tasks that would be too difficult/time consuming for one person alone
    • Label images for computer vision
    • Compensate people to create data sets for AI training

• Humans-as-a-service
  • Hundreds of thousands of people perform cheap labor hidden behind an online platform
  • Command people as if they’re operating a mindless machine
  • Perform click work, answer surveys, transcribing audio files
  • Median wage of $2 an hour

Photo: Brett Wallace
More reading that may be of interest!

• Are You Unwittingly Helping to Train Google’s AI Models? — Towards Data Science
• By typing captcha, you are actually helping AI’s training — AP News
• Potemkin AI — Real Life
• The Internet Is Enabling a New Kind of Poorly Paid Hell — The Atlantic
• The Ghost Workers in the Machine — Jacobin
• The rise of pseudo-AI: how tech firms quietly use humans to do bots’ work — The Guardian