"WE STRIVE TO MAKE COMPUTER SCIENCE AT BROWN UNIVERSITY A MORE RECEPTIVE AND INCLUSIVE SPACE FOR UNDERREPRESENTED RACIAL MINORITIES."

MOSAIC+

Join our Slack Channel:

Join our Email List:

1

Out in CS: Mentorship Program

A new initiative building LGBTQIA+ community in CS

Mentorship

1-on-1 or peer group advising from LGBTQIA+ upper-year mentors - from internship advice to navigating LGBTQIA+ student life

Sign up now at: http://tinyurl.com/brownoutincs form closes **Jan 31st**

How are people matched?

We pair mentors and mentees based on interest and identity preferences in the form. We especially encourage TGNC folks, femmes, and POC to join!

Community Events

- LGBTQIA+ Peer Connections
- Industry Panelists
- Game Nights
- + Submit your own ideas!

Questions? Feedback?

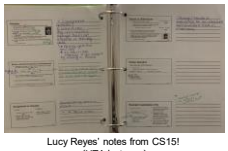
contact us at evan_dong@brown.edu and marcus_mitchell@brown.edu

*we welcome closeted & questioning folks - confidentiality is a top priority

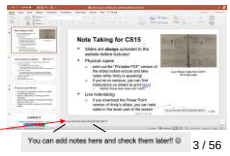
2

Note Taking for CS15

- Slides are **always** uploaded to the website before lectures!
- Physical copies
  - print out the "Printable PDF" version of the slides before lecture and take notes while Andy is speaking!
  - if you're on campus, you can find instructions on where to print here!
    - MyPrint Pickup desk closed until 1/25/21
- Live note-taking
  - if you download the Power Point version of Andy's slides, you can take notes in the lower part of the screen



Lucy Reyes' notes from CS15! (HTA last year)



3 / 56

3

Lecture Question Logistics

Google Forms

- Lecture Questions will be conducted through Google Forms
- Synchronous Students:
  - One Google Form per Lecture Question
  - Forms will be released in the zoom chat window
  - We will collect results real-time and discuss the answers during lecture
- Asynchronous Students:
  - One Google Form per lecture, containing all relevant questions
  - Forms will be released with the lecture recordings on the website
  - You will have 48 hours from the end of lecture to complete the form
- Graded on completion
- 5% of total grade
- Drop the lowest 4 quiz scores

4

Previous Lecture Review

- We model the “application world” as a system of collaborating objects
- Objects collaborate by sending each other messages
- Objects have properties and behaviors (things they know how to do)
- Objects typically composed of component objects

5

Lecture 2

Calling and Defining Methods in Java

6

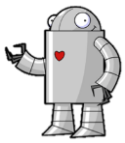
Outline

- Calling methods
- Declaring and defining a class
- Instances of a class
- Defining methods
- The `this` keyword

7

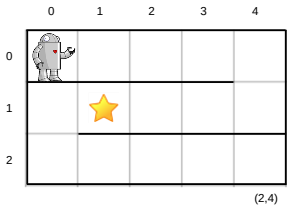
Meet samBot (kudos to former HTA Sam Squires)

- `samBot` is a robot who lives in a 2D grid world
- She knows how to do two things:
  - move forward any number of steps
  - turn right 90º
- We will learn how to communicate with `samBot` using Java



8

samBot's World

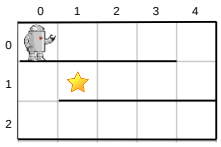


- This is `samBot`'s world
- `samBot` starts in the square at (0,0)
- She wants to get to the square at (1,1)
- Thick black lines are walls `samBot` can't pass through

9

Giving Instructions (1/3)

- **Goal**: move `samBot` from starting position to destination by giving her a list of instructions
- `samBot` only knows how to "move forward *n* steps" and "turn right"
- What instructions should be given?

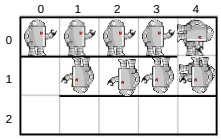


10

Giving Instructions (2/3)

Note: samBot moves in the direction her outstretched arm is pointing.
Yes, she can move sideways and upside down in this 2D world!

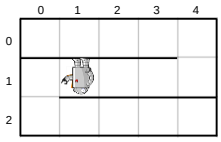
- "Move forward 4 steps."
- "Turn right."
- "Move forward 1 step."
- "Turn right."
- "Move forward 3 steps."



11

Giving Instructions (3/3)

- Instructions have to be given in a language `samBot` knows
- That's where Java comes in!
- In Java, give instructions to an object by **giving it commands**



12

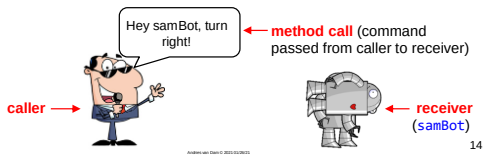
"Calling Methods": Giving Commands in Java (1/2)

- `samBot` can only handle commands she knows how to respond to
- These responses are called **methods!**
  - "method" is short for "method for responding to a command". Therefore, whenever `samBot` gets a command, she can respond by utilizing a predefined method
- Objects cooperate by giving each other commands
  - **caller** is the object giving the command
  - **receiver** is the object receiving the command

13

"Calling Methods": Giving Commands in Java (2/2)

- `samBot` already has one method for "move forward *n* steps" and another method for "turn right"
- When we send a command to `samBot` to "move forward" or "turn right" in Java, we are **calling a method on** `samBot`



14

Turning samBot right

Names don't have spaces! Style guide has capitalization conventions, e.g., camelCase

- `samBot`'s "turn right" method is called `turnRight`
- To call the `turnRight` method on `samBot`:

```
samBot.turnRight();
```

- To call methods on `samBot` in Java, you need to address her by name!
- Every command to `samBot` takes the form:

You can substitute anything in < >!

```
samBot.<method name(…)>;
```

; ends Java statement

- What are those parentheses at the end of the method for?

15

Moving samBot forward

- Remember: when telling `samBot` to move forward, you need to tell her how many steps to move
- `samBot`'s "move forward" method is named `moveForward`
- To **call** this method in Java:

```
samBot.moveForward(<number of steps>);
```

- This means that if we want her to move forward 2 steps, we say:

```
samBot.moveForward(2);
```


16

Calling Methods: Important Points

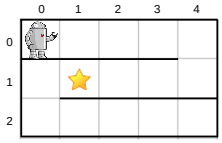
- Method calls in Java have parentheses after the method's name
- In the definition of the method, extra pieces of information to be passed into the method are called **parameters**; in the call to the method, the actual values passed in are called **arguments**
  - e.g. : in defining f(x), x is the parameter; in calling f(2), 2 is the argument
  - more on parameters and arguments next lecture!
- If the method needs any information, include it between the parentheses (e.g., `samBot.moveForward(2);`)
- If no extra information is needed, just leave the parentheses empty (e.g., `samBot.turnRight();`)

17

Guiding samBot in Java

- Tell `samBot` to move forward 4 steps → `samBot.moveForward(4);`
- Tell `samBot` to turn right → `samBot.turnRight();`
- Tell `samBot` to move forward 1 step → `samBot.moveForward(1);`
- Tell `samBot` to turn right → `samBot.turnRight();`
- Tell `samBot` to move forward 3 steps → `samBot.moveForward(3);`

"pseudocode"



Java code

18

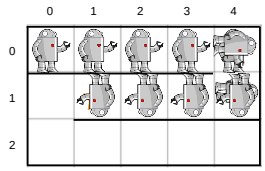
Hand Simulation

- Simulating lines of code **by hand** checks that each line produces correct action
  - we did this in slide 10 for pseudocode
- In **hand simulation**, **you play the role of the computer**
  - lines of code are "instructions" for the computer
  - try to follow "instructions" and see if you get desired result
  - if result is incorrect:
    - one or more instructions or the order of instructions may be incorrect

19

Hand Simulation of This Code

```
samBot.moveForward(4);
samBot.turnRight();
samBot.moveForward(1);
samBot.turnRight();
samBot.moveForward(3);
```



20

About Lecture Questions

- Increase engagement during lecture!
- Allow you to gauge your understanding of important concepts throughout lecture
- Give you participation points for paying attention during

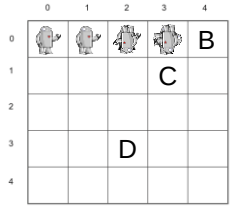
- Lecture questions are worth 5% of your grade! (See course missive)

21

Lecture Question

Where will `samBot` end up when this code is executed?

```
samBot.moveForward(3);
samBot.turnRight();
samBot.turnRight();
samBot.moveForward(1);
```



Choose one of the positions or
E: None of the above

22

Putting Code Fragments in a Real Program (1/2)

- Let's demonstrate this code for real
- First, put it inside real Java program
- Grayed-out code specifies context in which an arbitrary robot named `myRobot`, a parameter of the `moveRobot` method, executes instructions
  - part of **stencil code** written for you by the TAs, which also includes any robot's capability to respond to `moveForward` and `turnRight`– more on this later

```
public class RobotMover {

    /* additional stencil code elided*/

    public void moveRobot(Robot myRobot) {
        myRobot.moveForward(4);
        myRobot.turnRight();
        myRobot.moveForward(1);
        myRobot.turnRight();
        myRobot.moveForward(3);
    }
}
```

comment

23

Putting Code Fragments in a Real Program (2/2)

- Before, we've talked about objects that handle messages with "methods"
- Introducing a new concept… **classes**!

We're about to explain this part of the code!

```
public class RobotMover {

    /* additional code elided */

    public void moveRobot(Robot myRobot) {
        myRobot.moveForward(4);
        myRobot.turnRight();
        myRobot.moveForward(1);
        myRobot.turnRight();
        myRobot.moveForward(3);
    }
}
```


24

What is a class?

- A **class** is a blueprint for a certain type of object
- An object's class defines its properties and capabilities (methods)
  - more on this in a few slides!
- Let's embed the moveRobot code fragment (method) that moves `samBot` (or any other `Robot`) in a new class called `RobotMover`
- Need to tell Java compiler about `RobotMover` before we can use it

```
public class RobotMover {

    /* additional code elided */

    public void moveRobot(Robot myRobot) {
          myRobot.moveForward(4);
          myRobot.turnRight();
          myRobot.moveForward(1);
          myRobot.turnRight();
          myRobot.moveForward(3);
    }
}
```


25

Declaring and Defining a Class (1/3)

- Like a dictionary entry, first **declare** term, then provide **definition**
- First line **declares** `RobotMover` class
- Breaking it down:
  - `public` indicates any other object can use instances of this class
  - `class` indicates to Java compiler that we are about to define a new class
  - `RobotMover` is the name we have chosen for our class

declaration of the `RobotMover` class

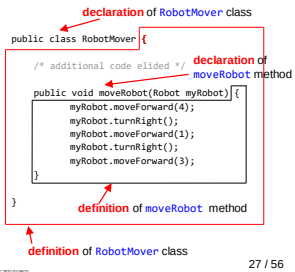
```
public class RobotMover {

    /* additional code elided */

    public void moveRobot(Robot myRobot) {
          myRobot.moveForward(4);
          myRobot.turnRight();
          myRobot.moveForward(1);
          myRobot.turnRight();
          myRobot.moveForward(3);
    }
}
```

Note: `public` and `class` are Java "reserved words" aka "keywords" and have pre-defined meanings in Java; use Java keywords a lot in the future

26

Declaring and Defining a Class (2/3)

- **Class definition** (aka "body") defines properties and capabilities of class
  - it is contained within curly braces that follow the class declaration
- A class's capabilities ("what it knows how to do") are defined by its **methods** – `RobotMover` thus far only shows one specific method, `moveRobot`
  - A method is a declaration followed by its body (also enclosed in {…} braces)
- A class's properties are defined by its **instance variables** – more on this next week



27

Declaring and Defining a Class (3/3)

- **General form for a class:**

```
<visibility> class <name> {          } declaration

    <code (properties and
capabilities) that defines class>    } definition
}
```

- to make code more compact, typically put opening brace on same line as declaration -- Java compiler doesn't care
- Each class goes in its own file, where **name of file matches name of class**
  - `RobotMover` class is contained in file "RobotMover.java"

28

The `Robot` class (defined by the TAs)

Note: Normally, support code is a "black box" that you can't examine


```
public class Robot {
    public void turnRight() {
        // code that turns robot right
    }

    public void moveForward(int numberOfSteps) {
        // code that moves robot forward
    }

    /* other code elided-- if you're curious, check out
    Robot.java in the stencil code!*/
}
```

in-line comment

- `public class Robot` **declares** a **class** called `Robot`
- Information about the properties and capabilities of `Robot`s (the **class definition**) goes within the red curly braces

29

Methods of the TA's `Robot` class

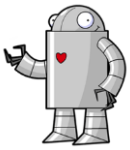
```
public class Robot {

    public void turnRight() {
        // code that turns robot right
    }

    public void moveForward(int numberOfSteps) {
        // code that moves robot forward
    }

    /* other code elided-- if you're curious, check
    out Robot.java in the stencil code!*/
}
```

- `public void turnRight()` and `public void moveForward(int numberOfSteps)` each **declare a method**
  - more on `void` later!
- `moveForward` needs to know how many steps to move, so the parameter is `int numberOfSteps` within parentheses
  - `int` tells compiler this parameter is an "integer" ("`moveForward` takes a single parameter called `numberOfSteps` of type `int`")

30

Classes and Instances (1/4)

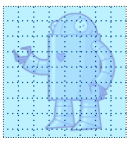
- `samBot` is an **instance** of class `Robot`
  - this means `samBot` is a particular `Robot` that was built using the `Robot` class as a blueprint (another **instance** could be `celyBot`)
- All `Robots` (all **instances** of the class `Robot`) have **the exact same capabilities**: the methods defined in the `Robot` class. What one `Robot` **instance** can do, they all can do since they are made with the same blueprint!
- All `Robot`s also have **the exact same properties** (i.e., every `Robot` has a `Color` and a `Size`)
  - they all have these properties but the values of these properties may differ between instances (e.g., a big `samBot` and small `celyBot`)



Cely from Love Island season 2

31

Classes and Instances (2/4)

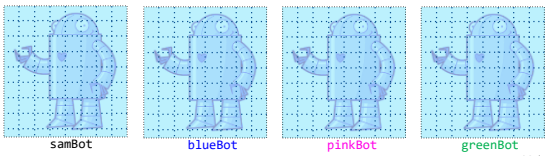


The `Robot` class is like a blueprint

32

Classes and Instances (3/4)

We can use the `Robot` class to build actual `Robot`s - **instances** of the class `Robot`, whose properties (like their color in this case) may vary (next lecture)

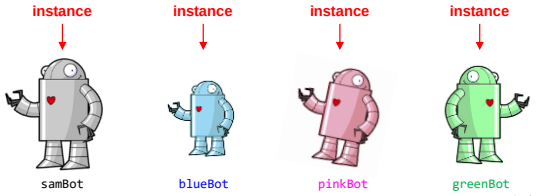


`samBot` `blueBot` `pinkBot` `greenBot`

33

Classes and Instances (4/4)

Method calls are done on instances of the class. These are four instances of the same class (blueprint).



34

Lecture Question

You know that `blueBot` and `pinkBot` are instances of the same class. Let's say that the call `pinkBot.chaChaSlide();` makes `pinkBot` do the cha-cha slide. Which of the following is true?

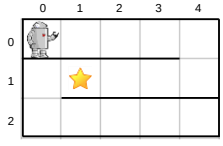
A. The call `blueBot.chaChaSlide();` might make `blueBot` do the cha-cha slide or another popular line dance instead

B. The call `blueBot.chaChaSlide();` will make `blueBot` do the cha-cha slide

C. You have no guarantee that `blueBot` has the method `chaChaSlide();`

35

Defining Methods



- We have already learned about **defining classes**, let's now talk about **defining methods**
- Let's use a variation of our previous example

```
public class RobotMover {
    /* additional code elided */

    public void moveRobot(Robot myRobot) {
        // Your code goes here!
        // …
        // …
    }
}
```


36

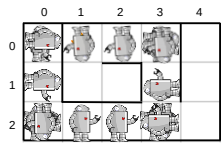
Declaring vs. Defining Methods

- **Declaring** a method means the class knows how to do some task, like `pinkBot` can `chaChaSlide()`
- **Defining** a method actually explains how the class executes this task (i.e. what sequence of commands it specifies)
  - `chaChaSlide()` could include: stepping backwards, alternating feet, stepping forward
- Usually, you will need to both **define** and **declare** your methods

37 / 56

37

A Variation on `moveRobot` (1/2)




```
public class RobotMover {
    /* additional code elided */

    public void moveRobot(Robot myRobot) {
        myRobot.turnRight();
        myRobot.moveForward(2);
        myRobot.turnRight();
        myRobot.turnRight();
        myRobot.turnRight();
        myRobot.moveForward(3);
        myRobot.turnRight();
        myRobot.turnRight();
        myRobot.turnRight();
        myRobot.moveForward(2);
        myRobot.turnRight();
        myRobot.turnRight();
        myRobot.turnRight();
        myRobot.moveForward(2);
    }
}
```


38 / 56

38

A Variation on `moveRobot` (2/2)

- Lots of code for a simple problem.
- `samBot` only knows how to turn right, so have to call `turnRight` three times to make her turn left
- If she understood how to "turn left", would be much less code!
- We can ask the TAs to modify `samBot` to turn left by **declaring** and **defining a method** called `turnLeft`

```
public class RobotMover {
    /* additional code elided */

    public void moveRobot(Robot myRobot) {
        myRobot.turnRight();
        myRobot.moveForward(2);
        myRobot.turnRight();
        myRobot.turnRight();
        myRobot.turnRight();      } "turn left"
        myRobot.moveForward(3);
        myRobot.turnRight();
        myRobot.turnRight();
        myRobot.turnRight();      } "turn left"
        myRobot.moveForward(2);
        myRobot.turnRight();
        myRobot.turnRight();
        myRobot.turnRight();      } "turn left"
        myRobot.moveForward(2);
    }
}
```


39 / 56

39

Defining a Method (1/2)

- Almost all methods take on this general form:

explanation in later lecture

```
<visibility> <type> <name> (<parameters>) {
    <list of statements within method>
}
```

- When **calling** `turnRight` or `moveForward` on an **instance** of the `Robot` class, all code between method's curly braces is executed

```
public class Robot {

    public void turnRight() {
        // code that turns robot right
    }

    public void moveForward(int numberOfSteps) {
        // code that moves robot forward
    }

}
```


40

Defining a Method (2/2)

- We're going to **define** a new method: `turnLeft`
- To make a `Robot` turn left, tell it to turn right three times

```
public class Robot {

    public void turnRight() {
        // code that turns robot right
    }

    public void moveForward(int numberOfSteps) {
        // code that moves robot forward
    }

    public void turnLeft() {
        this.turnRight();
        this.turnRight();
        this.turnRight();
    }
}
```


41

The `this` keyword (1/2)

- When working with `RobotMover`, we were talking to `samBot`, an instance of class `Robot`
- To tell her to turn right, we said "`samBot.turnRight();`"
- Why do the TAs now write "`this.turnRight();`"?

```
public class Robot {

    public void turnRight() {
        // code that turns robot right
    }

    public void moveForward(int numberOfSteps) {
        // code that moves robot forward
    }

    public void turnLeft() {
        this.turnRight();
        this.turnRight();
        this.turnRight();
    }
}
```


42

The `this` keyword (2/2)

- The `this` keyword is how an instance (like `samBot`) can call a method on itself
- Use `this` to call a method of `Robot` class from within another method of the `Robot` class
- When `samBot` is told by, say, a `RobotMover` instance to `turnLeft`, she responds by telling herself to `turnRight` three times
- `this.turnRight();` means "hey me, turn right!"
- `this` is optional, but CS15 expects it

```
public class Robot {

    public void turnRight() {
        // code that turns robot right
    }

    public void moveForward(int numberOfSteps) {
        // code that moves robot forward
    }

    public void turnLeft() {
        this.turnRight();
        this.turnRight();
        this.turnRight();
    }
}
```


43 / 56

43

We're done!

- Now that `Robot` has `turnLeft()`, can call `turnLeft()` on any instance of `Robot`
- We will see how we can use `turnLeft()` to simplify our code in a few slides

```
public class Robot {

    public void turnRight() {
        // code that turns robot right
    }

    public void moveForward(int numberOfSteps) {
        // code that moves robot forward
    }

    public void turnLeft() {
        this.turnRight();
        this.turnRight();
        this.turnRight();
    }
}
```


44 / 56

44

Lecture Question

```
public class Robot {

    /* additional code elided */

    public void turnLeft() {
        this.turnRight();
        this.turnRight();
        this.turnRight();
    }
}
```

Given this method, what can we say about `this.turnRight()`?

A. Other objects cannot call the `turnRight()` method on instances of the `Robot` class

B. The current instance of the `Robot` class is calling `turnRight()` on another instance of `Robot`

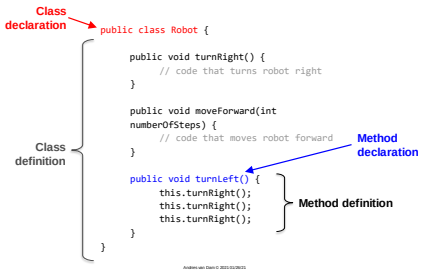
C. The current instance of the `Robot` class is calling the `turnRight()` method on itself

D. The call `this.turnRight();` will not appear anywhere else in the `Robot`'s class definition

45 / 56

45

Summary



46

Simplifying our code using `turnLeft`

```
public class RobotMover {
    public void moveRobot(Robot myRobot) {
        myRobot.turnRight();
        myRobot.moveForward(2);
        myRobot.turnRight();
        myRobot.turnRight();
        myRobot.turnRight();
        myRobot.moveForward(3);
        myRobot.turnRight();
        myRobot.turnRight();
        myRobot.turnRight();
        myRobot.moveForward(2);
        myRobot.turnRight();
        myRobot.turnRight();
        myRobot.turnRight();
        myRobot.moveForward(2);
    }
}
```

```
public class RobotMover {
    public void moveRobot(Robot myRobot) {
        myRobot.turnRight();
        myRobot.moveForward(2);
        myRobot.turnLeft();
        myRobot.moveForward(3);
        myRobot.turnLeft();
        myRobot.moveForward(2);
        myRobot.turnLeft();
        myRobot.moveForward(2);
    }
}
```

We've saved a lot of lines of code by using turnLeft!

This is good! More lines of code makes your program harder to read and more difficult to debug and maintain.

47

`turnAround` (1/3)

- The TAs could also define a method that turns the `Robot` around 180º.
- See if you can declare and define the method **`turnAround`**

```
public class Robot {
    public void turnRight() {
        // code that turns robot right
    }

    public void moveForward(int numberOfSteps) {
        // code that moves robot forward
    }

    public void turnLeft() {
        this.turnRight();
        this.turnRight();
        this.turnRight();
    }

    // your code goes here!
    // …
    // …
    // …
}
```


48

`turnAround` (2/3)

- Now that the `Robot` class has the method `turnAround`, we can call the method on any instance of the class `Robot`
- There are other ways of implementing this method that are just as correct

```
public class Robot {
      public void turnRight() {
            // code that turns robot right
      }

      public void moveForward(int numberOfSteps) {
            // code that moves robot forward
      }

      public void turnLeft() {
            this.turnRight();
            this.turnRight();
            this.turnRight();
      }

      public void turnAround() {
            this.turnRight();
            this.turnRight();
      }
}
```


49 / 56

49

`turnAround` (3/3)

- Instead of calling `turnRight`, could call our newly created method, `turnLeft`
- Both of these solutions are equally correct, in that they will turn the robot around 180º
- How do they differ? When we try each of these implementations with `samBot`, what will we see in each case?

```
public class Robot {
      public void turnRight() {
            // code that turns robot right
      }

      public void moveForward(int numberOfSteps) {
            // code that moves robot forward
      }

      public void turnLeft() {
            this.turnRight();
            this.turnRight();
            this.turnRight();
      }

      public void turnAround() {
            this.turnRight();
            this.turnRight();
      }
}
```


50 / 56

50

Summary (1/2)

- Classes
  - a **class** is a blueprint for a certain type of object
  - example: `Robot` is a class
- Instances
  - an **instance** of a class is a particular member of that class whose methods we can call
  - example: `samBot` is an **instance** of `Robot`

51 / 56

51

Summary (2/2)

- Calling methods
  - an instance can call on the methods defined by its class
  - **general form**: `instance.<method name>(<parameters>)`
  - example: `samBot.turnRight();`
- Defining methods
  - how we describe a capability of a class
  - **general form:** `<visibility> <type> <name> (<parameters>)`
  - example: `public void turnLeft() { … }`
- The `this` keyword
  - how an instance calls a method on itself within its class definition
  - example: `this.turnRight()`

52

Announcements

- Hello World out today!
  - Due Saturday 1/30
  - No Early or Late Hand-in
- Lab 0 Linux and Terminal wrapping up today
  - If you have not done Lab 0 contact the HTAs and sign up for a lab section ASAP
  - Review GitHub/IntelliJ setup before lab!

53

IT in the News

ft. Socially Responsible Computing!



54

Social Media & the Insurrection

- January 6, 2021: pro-Trump mob storms US Capitol
- Facebook
  - algorithms reward extreme views & conspiracy theories
  - emphasize groups for community → strengthens hate-based groups (unintended but permitted consequence)
    - private/secret groups largely unmoderated
  - pushed into minimal content moderation policy
- Twitter
  - extreme comments more popular (more likes/retweets)
  - exempted Trump & other politicians from content policies ("public interest exemption")

Facebook says private groups are its future. Some are hubs for misinformation and hate.

Social Media & the Insurrection

- Aftermath: Trump banned from
  - Twitter (permanent)
  - Facebook/Instagram (indefinite)
  - Twitch (permanent)
  - Shopify (permanent)
  - Stripe (permanent)
  - Snapchat
  - TikTok
  - YouTube
  - ...and more
- What took them so long?
- **engagement = profit**
- fear of conservative backlash
- reluctance to be labeled censor/publisher
- What **are** the responsibilities of a social media platform?

NBC news

Opinion: It took a mob riot for Twitter to finally ban Trump

January 8, 2021 LA Times opinion