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!
- 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.
Lecture Question Logistics

- 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

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

Lecture 2

Calling and Defining Methods in Java
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 90°
- We will learn how to communicate with samBot using Java

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
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?

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."

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
“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

- `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`

“Calling Methods”: Giving Commands in Java (2/2)

- `samBot`’s “turn right” method is called `turnRight`
- To call the `turnRight` method on `samBot`:
  ```java
  samBot.turnRight();
  ```
- To call methods on `samBot` in Java, you need to address her by name!
- Every command to `samBot` takes the form:
  ```java
  samBot.<method name(...)>
  ```
- What are those parentheses at the end of the method for?

Turning `samBot` right

- `samBot`’s “turn right” method is called `turnRight`
- To call the `turnRight` method on `samBot`:
  ```java
  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 `< >`
  ```java
  samBot.<method name(...)>
  ```
- What are those parentheses at the end of the method for?
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:
  ```java
  samBot.moveForward(<number of steps>);
  ```
- This means that if we want her to move forward 2 steps, we say:
  ```java
  samBot.moveForward(2);
  ```

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();`)!

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);`
Hand Simulation

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

Hand Simulation of This Code

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

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 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

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

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

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!

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

```java
public class RobotMover {
    /* additional code elided */
    public void moveRobot(Robot myRobot) {
        myRobot.moveForward(4);
        myRobot.turnRight();
        myRobot.moveForward(1);
        myRobot.turnRight();
        myRobot.moveForward(3);
    }
}
```

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

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.

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
Declaring and Defining a Class (3/3)

- General form for a class:
  ```
  \{"visibility\} class <name> {
  \<code (properties and capabilities) that defines class> \}
  \}
  ```

- 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"

The Robot class (defined by the TAs)

```java
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!*/
}
```

Methods of the TA's Robot class

- public void turnRight() and public void moveForward(int numberOfSteps) each declare a method
  - turnRight needs to know how many steps to turn, so the parameter is int numberOfSteps
  - moveForward needs to know how many steps to move, so the parameter is int numberOfSteps
  - moveForward takes a single parameter called numberOfSteps of type int
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 **Robots** 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**)

Classes and Instances (2/4)

- The **Robot** class is like a **blueprint**

Classes and Instances (3/4)

We can use the **Robot** class to build actual **Robots - instances** of the class **Robot**, whose properties (like their color in this case) may vary (next lecture)
Classes and Instances (4/4)

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

blueBot | pinkBot | greenBot

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();

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

```java
public class RobotMover {
    //= additional code elided /=
    public void moveRobot(Robot myRobot) {
        // Your code goes here!
    }
```
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

A Variation on `moveRobot` (1/2)

```java
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);
    }
}
```

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`
Defining a Method (1/2)

- Almost all methods take on this general form:

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

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

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

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

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();"`?
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.

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

### 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.

```java
public class Robot {
    /* additional code elided */
    public void turnLeft() {
        this.turnRight();
        this.turnRight();
        this.turnRight();
    }
}
```

### Lecture Question

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.
Summary

Class declaration

Class definition

Method declaration

Method definition

Simplifying our code using `turnLeft`

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.

turnAround (1/3)

- The TAs could also define a method that turns the `Robot` around 180°.
- See if you can declare and define the method `turnAround`
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();
    }
}
```

turnAround (3/3)

- Instead of calling `turnRight`, we could call our newly created method, `turnLeft`.
- Both of these solutions are equally correct, in that they will turn the robot around 180°.
- 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();
    }
}
```

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`
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();`

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!

IT in the News

*ft. Socially Responsible Computing!*
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’)

Aftermath: Trump banned from
- Twitter (permanent)
- Facebook/Instagram (indefinite)
- Twitch (permanent)
- Shopify (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?