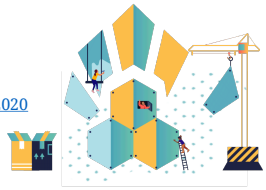
Hack@Brown 2020

Come to our info session:
https://tinyurl.com/info2020

And apply here:
https://tinyurl.com/hackatbrown2020
Due 9/13



WiCS

Women In Computer Science

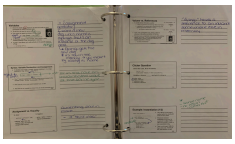
Dedicated to improving diversity and inclusion across gender identity in CS.

Join our listserv to get updates and hear opportunities!
Email: **wics@lists.cs.brown.edu**
Add yourself to the listserv: **https://tinyurl.com/brownuwics**

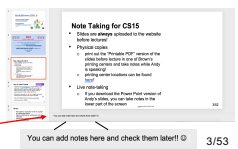
WELCOME BACK EVENT ft. Kabob & Curry
Thursday, 9/12/19
CIT 368

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 in one of Brown's printing centers and take notes while Andy is speaking!
  - printing center locations can be found 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



My own notes from CS15!



3/53

How to Install TopHat

- Computer: Go to https://tophat.com → Click *Signup* in upper right corner → select *Student* → join with course code or *Search by School* → input info under *Account* → enter your Banner ID under *Grading* → Add your phone number to submit responses in class via text under *Phone*
- IOS/Android: Download Top Hat Lecture App→ click *Create Student Account* and follow instructions to complete
- Link with Detailed Instructions: https://tinyurl.com/y6ythebb
- CS15 Course Code: 783865

4/53

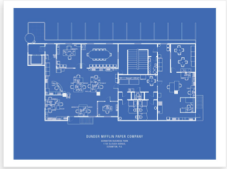
Review

- 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 are typically composed of component objects

5/53

Lecture 2

Calling and Defining Methods in Java



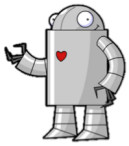
6/53

Outline

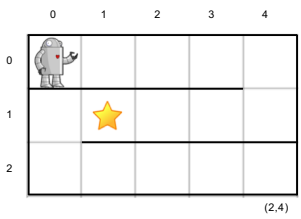
- Calling methods
- Declaring and defining a class
- Instances of a class
- Defining methods
- The `this` keyword

Meet samBot (kudos to former headTA 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



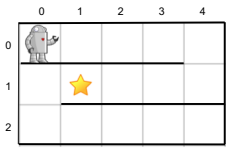
samBot's World



- 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)

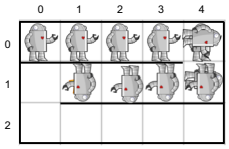
- **Goal**: move `samBot` from starting position to destination by giving her a list of instructions
- `samBot` only knows how to "move forward *n* steps" and "turn right"
- What instructions should be given?



Giving Instructions (2/3)

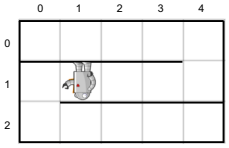
Note: samBot moves in the direction her outstretched arm is pointing. Yes, she can move sideways and upside down in this 2D world!

- "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**

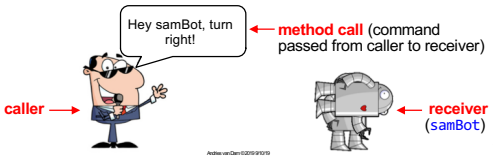


"Calling Methods": Giving Commands in Java (1/2)

- `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 method.
- Objects cooperate by giving each other commands
  - **caller** is the object giving the command
  - **receiver** is the object receiving the command

"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`.



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, need to address her by name!
- Every command to `samBot` takes the form:

```
samBot.<method name(…)>;
```

You can substitute anything in < >!

; ends Java statement

- 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:

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

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

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

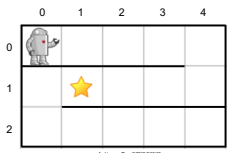

Calling Methods: Important Points

- 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);`

"pseudocode"



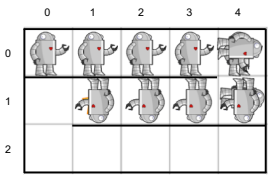
Java code

Hand Simulation

- Simulating lines of code **by hand** checks that each line produces correct action
  - we did this in slide 7 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

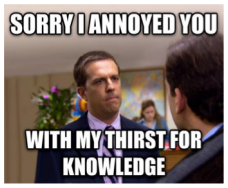
Hand Simulation of This Code

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



About TopHat Questions

- Increase engagement during lecture!
- We encourage working with a neighbor and discussing concepts on all TopHat questions
- Can use app, website
  - If you need an device to access TopHat, you can borrow a laptop from the IT Service Center on 5th floor of Page-Robinson Hall.

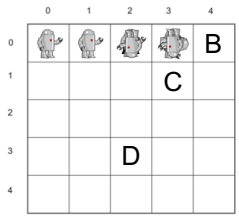


- TopHat questions are worth 5% of your grade! (See course missive)

TopHat 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

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 our robot named `samBot` executes instructions
  - it is part of the **stencil code** written for you by the TAs, which also includes `samBot`'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

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);
    }
}
```

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` instance) 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);
    }
}
```


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

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);
    }
}
```


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

```
public class RobotMover {

    /* additional code elided */

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

definition of `moveRobot` method

definition of `RobotMover` class

Declaring and Defining a Class (3/3)

- **General form for a class:**

```
<visibility> class <name> {

    <code (properties and
capabilities) that defines class>

}
```

declaration

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"

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

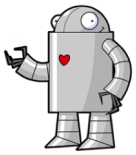
Methods of the TA's `Robot` class

```
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!*/
}
```

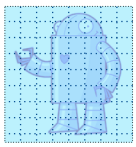
- `public void turnRight()` and `public void moveForward(int numberOfSteps)` each **declare a method**
  - more on `void` later!
- `moveForward` needs to know how many steps to move, so the parameter is `int numberOfSteps` within parentheses
  - `int` tells compiler this parameter is an "integer" (we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



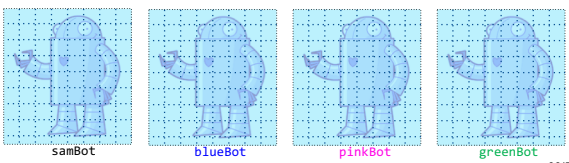
Classes and Instances (2/4)



The `Robot` class is like a blueprint

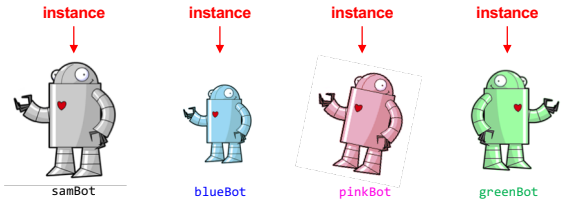
Classes and Instances (3/4)

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).



34/53

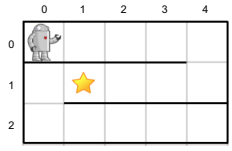
TopHat Question

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();` will make `blueBot` do the cha-cha slide
B. The call `blueBot.chaChaSlide();` might make `blueBot` do the cha-cha slide or another popular line dance instead
C. You have no guarantee that `blueBot` has the method `chaChaSlide();`

35/53

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

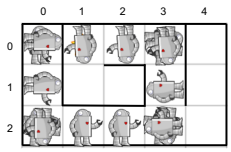
```
public class RobotMover {
    /* additional code elided */

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

36/53

Declaring vs. Defining Methods

- **Declaring** a method says the class knows how to do some task like `pinkBot` can `chaChaSlide()`
- **Defining** a method actually explains how the class completes this task (what command it gives) `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)




```
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);
    }
}
```


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();    > "turn left"
        myRobot.turnRight();   /
        myRobot.moveForward(3);
        myRobot.turnRight();   \
        myRobot.turnRight();    > "turn left"
        myRobot.turnRight();   /
        myRobot.moveForward(2);
        myRobot.turnRight();   \
        myRobot.turnRight();    > "turn left"
        myRobot.turnRight();   /
        myRobot.moveForward(2);
    }
}
```

Defining a Method (1/2)

```
public class Robot {

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

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


}
```

- 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

Defining a Method (2/2)

```
public class Robot {

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

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

    public void turnLeft() {
        //The TA's code goes here!!
        //Here you'll have the method definition!
    }
}
```

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

The `this` keyword (1/2)

```
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 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)

```
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 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

We're done!

```
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();
     }
}
```

- Have now seen our first method definition!
- Now that `Robot` has `turnLeft`, can call `turnLeft` on any instance of `Robot`

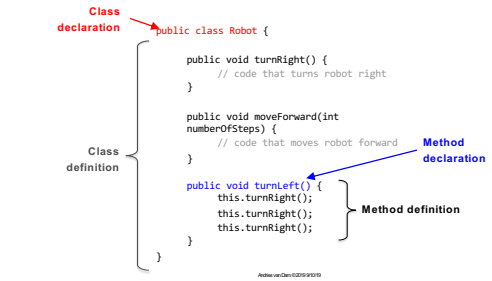
TopHat Question

```
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

Summary



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.

`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!
    // …
    // …
    // …
}
```

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`, 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

- HW1 is out!
- Sign up on piazza!! Link on website!
- Sections start today
  - you should have a section by now – if not, email the Head TAs ASAP (cs0150headtas@lists.brown.edu)
  - if you try to attend a section you aren't signed up for, you will not get checked off
  - find assigned room in the same link that you used to sign up
- For the best email response time: email the TA listserv! (cs0150tas@lists.brown.edu)
  - next best: email cs0150headtas
  - slow response: email individual TA – don't do it!