Lecture 2

Calling and Defining Methods in Java
WHO WE ARE:
A group of volunteers that writes and teaches original interactive STEM lessons to local high schoolers.

WHEN WE DO THAT:
Every Saturday from 10:30am-1pm

OUR GOAL:
To get high school students excited about learning STEM through experiments, hands-on demonstrations, and applicable learning.

EXAMPLES OF PAST LESSONS:
- Dogfish Shark Dissection
- Physics of Black Holes
- Science of Sleep
- Math in Nature
-- Exercise Physiology
- Intro to Computer Science

FOR MORE INFORMATION:
brown.edu/academics/science-center/outreach/science-prep/

TO APPLY TO JOIN US:
BSPMentorPortal.weebly.com

APPS DUE THURSDAY SEPT 15TH @ 11:59PM
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:
  o move forward any number of steps
  o 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 that *samBot* can’t pass through
Giving Instructions (1/3)

• **Goal**: move *samBot* from her starting position to her 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 commands`.
“Calling Methods”: Giving Commands in Java (1/2)

• **samBot** can only handle commands that she knows how to respond to

• These responses are called **methods**!
  - “method” is short for “method for responding to a command”

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

The method call (command passed from caller to receiver)

The caller

The receiver

(samBot)
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, need to address her by name!
- Every command to `samBot` takes the form:
  
  ```java
  samBot.<method name(...)>
  ```
  
  You substitute for 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, 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, say:

```
samBot.moveForward(2);
```
Calling Methods: Important Points

- Method calls in Java have parentheses after the method’s name.
- Extra pieces of information passed to a method are called **parameters**; the actual values passed in are called **arguments**.
  - e.g.: in defining \( f(x) \), \( x \) is the parameter; in using \( f(2) \), 2 is the argument.
  - more on this next lecture!
- If the method needs any information, include it between parentheses (e.g., \( \text{samBot.moveForward}(2) \)).
- If no extra information is needed, just leave the parentheses empty (e.g., \( \text{samBot.turnRight}() \)).
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 of This Code

```java
samBot.moveForward(4);
samBot.turnRight();
samBot.moveForward(1);
samBot.turnRight();
samBot.moveForward(3);
```
Clicker Question

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

```java
samBot.moveForward(3);
samBot.turnRight();
samBot.turnRight();
samBot.moveForward(1);
```
Putting Code Fragment 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 `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

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

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

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

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

We’re about to explain this part of the code!
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!
- So far, we’ve been working within the class `RobotMover`
- We need to tell Java about our `RobotMover`

```java
public class RobotMover {

    /* additional code elided */

    public void moveRobot(Robot samBot) {
        samBot.moveForward(4);
        samBot.turnRight();
        samBot.moveForward(1);
        samBot.turnRight();
        samBot.moveForward(3);
    }
}
```
Declaring and Defining a Class (1/3)

• As with dictionary entry, first declare term, then provide definition

• First line declares RobotMover class

• Breaking it down:
  o public indicates that anyone can use this class
  o class indicates to Java that we are about to define a new class
  o RobotMover is the name that we have chosen for our class

\[
\text{public class RobotMover} \{
    /* additional code elided */
    \text{public void moveRobot(Robot samBot)} \{
        samBot.moveForward(4);
        samBot.turnRight();
        samBot.moveForward(1);
        samBot.turnRight();
        samBot.moveForward(3);
    \}
\}
\]

Note: public and class are Java “reserved words” aka “keywords” and have pre-defined meanings in Java; we’ll be using 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 knows this very specific **moveRobot** method

- A class’s properties are defined by its **instance variables** – more on this next week

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

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

**definition** of the *RobotMover* class
Declaring and Defining a Class (3/3)

- General form for a class:

  `<visibility> class <name> {`

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

  `}`

- 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

```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!*/
}
```

- public class Robot declares a class called **Robot**
- Information about the properties and capabilities of **Robots** (the class definition) goes within the 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
  o more on void later!

• Since moveForward needs to know how many steps to move, put int numberOfSteps within parentheses
  o int tells Java this parameter is an “integer” (we say “moveForward takes a single parameter called numberOfSteps of type integer”)

Andries van Dam  2015 09/13/16
Classes and Instances (1/4)

• We’ve been saying *samBot* is a *Robot*

• We’ll now refer to her as an *instance* of class *Robot*
  
  o this means *samBot* is a particular *Robot* built using
    *Robot* class as a blueprint

• All *Robots* (all *instances* of the class *Robot*) have the exact same capabilities: the methods defined in the *Robot* class

• All *Robots* also have the same properties (i.e. every *Robot* has a *Color* and a *Size*)
  
  o values of these properties may differ between instances
    (e.g. we could have a blue *Robot* and red *Robot*)
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 may vary (next lecture)
Classes and Instances (4/4)

Method calls are done on instances of the class

- **instance**
  - samBot
- **instance**
  - blueBot
- **instance**
  - pinkBot
- **instance**
  - greenBot
Clicker 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();`
Defining Methods

- Now that we know about calling methods, classes and instances, let’s talk about **defining methods**

- To understand this, let’s use a variation of our previous example

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

    public void moveRobot(Robot samBot) {
        // Your code goes here!
        // ...
        // ...
    }
}
```
Declaring vs. Defining Methods

- Declaring a method says the class knows how to do some task

- Defining a method actually explains how the class completes this task (what command it gives)

- Usually you will need to both define and declare your methods
A variation

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

    public void moveRobot(Robot samBot) {
        samBot.turnRight();
        samBot.moveForward(2);
        samBot.turnRight();
        samBot.turnRight();
        samBot.turnRight();
        samBot.moveForward(3);
        samBot.turnRight();
        samBot.turnRight();
        samBot.turnRight();
        samBot.moveForward(2);
        samBot.turnRight();
        samBot.turnRight();
        samBot.moveForward(2);
    }
}
```
A variation

• 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 simpler!

• We can ask the TAs to modify *samBot* to turn left by defining a method called *turnLeft*
Defining a Method (1/2)

Almost all methods take on this general form:

```
<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)

```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() {
        // The TA’s code goes here!!
    }
}
```

- Adding a new method: `turnLeft`
- To make a `Robot` turn left, tell her to turn right three times
The **this** keyword (1/2)

```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 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();
    }

}
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();
        this.turnRight();
    }

}
Clicker Question

Give 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

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

}
Simplifying our code using `turnLeft`

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

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

We’ve saved a lot of lines of code by using `turnLeft`!
• The TAs could also define a method that turns the Robot around 180°.
• See if you can declare and define the method `turnAround`

```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();
    }

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


**turnAround**

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

```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();
    }

    public void turnAround() {
        this.turnRight();
        this.turnRight();
    }
}
```
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?
Summary (1/2)

- Classes
  - A class is a blueprint for a certain type of object

- Instances
  - An instance of a class is a particular member of that class, on which we can call methods.
  - Example: samBot is an instance of Robot
Summary (2/2)

- Calling methods
  - an object sends a message to another object
  - **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
  - example: `this.turnRight()`
Announcements (1/2)

- Hack@Brown has 2 information sessions
  - Tuesday, 9/13 at 8pm in BERT 130 and Wednesday, 9/14 at 6:30pm in RISD's Graphic Design Gallery
  - Learn more about different teams that help organize Hack@Brown!
  - Outreach, Design, Dev, Sponsorship, Food & Logistics, Hardware, Media, Workshops, Experience
- Hours start today at 4:30 – see website for full schedule
- Labs start today
  - you should have a lab section by now – if not, email the Head TAs ASAP
  - if you try to attend a lab section you aren’t signed up for, you will not get checked off
Announcements (2/2)

• If you haven’t created a REEF account, please do so by Thursday before class. Instructions were sent out in an email.
• For the best email response time: email the cs015tas
  o next best: email cs015headtas
  o slow response: email an individual TA – don’t do it!!