Lecture 2
Calling and Defining Methods in Java
Outline

• Calling methods
• Declaring and defining a class
• Instances of a class
• Defining methods
• The this keyword
Meet samBot  (kudos to former head TA 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

- **Goal**: move samBot from her starting position to her destination by giving her a list of instructions
- samBot only knows instructions “move forward n steps” and “turn right”
- What instructions should we give her?
Giving Instructions

Note: samBot moves in the direction her outstretched arm is pointing; yes, she can move upside down in this 2D world

- “Move forward 4 steps.”
- “Turn right.”
- “Move forward 1 step.”
- “Turn right.”
- “Move forward 3 steps.”
Giving Instructions

• Have to give samBot these instructions in a language she understands
• That’s where Java comes in!
• In Java, give instructions to an object by sending it messages
“Calling Methods”: Sending Messages in Java

• samBot can only handle messages that she knows how to respond to
• These responses are called **methods**!
  o “method” is short for “method for responding to a message”
• Objects cooperate by sending each other messages.
  o object sending message is the **caller**
  o object receiving message is the **receiver**
“Calling Methods”: Sending Messages in Java

- samBot already has one method for “move forward n steps” and another method for “turn right”
- When we send a message to samBot to “move forward” or “turn right” in Java, we are calling a method on samBot.

The caller

Hey samBot, turn right!

The method call (message passed from caller to receiver)

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 `< >`!
- What are those parentheses at the end of the method for?

; ends Java statement
Moving samBot forward

• 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:
  ```java
  samBot.moveForward(<number of steps>);
  ```
• This means that if we want her to move forward 2 steps, say:
  ```java
  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
  o e.g. : in defining f(x), x is the parameter; in using f(2), 2 is the argument
  o more on this next lecture!

• If the method needs any information, include it between 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 of This Code

```java
samBot.moveForward(4);
samBot.turnRight();
samBot.moveForward(1);
samBot.turnRight();
samBot.moveForward(3);
```
Putting Code Fragment in a Real Program

• Let’s demonstrate this code for real
• First, need to put it inside real Java program
• Grayed-out code specifies context in which `samBot` executes these instructions
  o 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

- 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)
• 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

```java
declaration of the RobotMover class
public class RobotMover {
    /* additional code elided */
    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:

\[
\begin{align*}
\langle \text{visibility} \rangle & \quad \text{class} & \langle \text{name} \rangle & \{
\langle \text{code (properties and capabilities) that defines class} \rangle \\
\}\quad \text{definition}
\end{align*}
\]

• Each class goes in its own file, where name of file matches name of class

  o 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

```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 void turnRight()` and `public void moveForward(int numberOfSteps)` each declare a method
- Since `moveForward` needs to know how many steps to move, we put `int numberOfSteps` within the parentheses
  - `int` is Java’s way of saying this parameter is an “integer” (we say “of type integer”)
Classes and Instances (1/3)

- We’ve been saying `samBot` is a `Robot`
- We’ll now refer to her as an instance of class `Robot`
  - 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
Classes and Instances (2/3)

The Robot class is like a blueprint
Classes and Instances (3/3)

We can use the Robot class to build actual Robots - instances of the class Robot, whose properties may vary (next lecture)
Classes and Instances

Method calls are done on instances of the class

instance

- samBot
- blueBot
- pinkBot
- greenBot
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!
        // ...
        // ...
    }
}
```
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.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`

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

    <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!!
    }
}

• 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!

```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();
        this.turnRight();
    }
}
```

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

Summary

Class declaration

Method declaration

Method definition
Simplifying our code using `turnLeft`

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

• The TAs could also define a method that turns the Robot around 180°.
• 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!
    // ...
    // ...
    // ...
}

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!   
    // ...
    // ...
    // ...
}

Andries van Dam © 2015 09/15/15
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.
turnAround

• 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?

```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();
        this.turnRight();
    }
}
```
Summary (1/2)

• Classes
  o a **class** is a blueprint for a certain type of object

• Instances
  o an **instance** of a class is a particular member of that class, on which we can call methods.
  o example: **samBot** is an **instance** of **Robot**
Summary (2/2)

• Calling methods
  o an object sends a message to another object
  o general form: `instance.<method name>(<parameters>)`
  o example: `samBot.turnRight();`

• Defining methods
  o how we describe a capability of a class
  o example: `public void turnLeft() { ... }`

• The `this` keyword
  o how an instance calls a method on itself
  o example: `this.turnRight()`
Course Advertisement

- IT1340B. The Panorama and 19th-century Visual Culture
  - uses Touch Art Gallery and Brown YURT - Ultimate Reality Theatre to explore Garibaldi panorama and more

- Meets TTH 1:00-2:20, Digital Scholarly Lab
Announcements

- Hours start today at 4:30 – see website for full schedule

- Labs start tonight
  - 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

- For the best email response time: email the cs015tas
  - next best: email cs015headtas
  - slow response: email an individual TA -- don’t do it!!