




Due Monday by 11:59pm **Points** 2 **Available** Feb 8 at 9pm - Feb 13 at 11:59pm

Lab 4 — More Interfaces and Inheritance



-  **Home** (<https://northeastern.instructure.com/courses/136715/pages/home>)
-  **Modules** (<https://northeastern.instructure.com/courses/136715/modules>)



(<https://northeastern.instructure.com/courses/136715/pages/home>) (<https://northeastern.instructure.com/courses/136715/modules>) (<https://northeastern.instructure.com/courses/136715/pages/home>) (<https://northeastern.instructure.com/courses/136715/modules>) (<https://northeastern.instructure.com/courses/136715/pages/home>) (<https://northeastern.instructure.com/courses/136715/modules>) (<https://northeastern.instructure.com/courses/136715/pages/home>) (<https://northeastern.instructure.com/courses/136715/modules>) (<https://northeastern.instructure.com/courses/136715/pages/home>) (<https://northeastern.instructure.com/courses/136715/modules>) (<https://northeastern.instructure.com/courses/136715/pages/home>) (<https://northeastern.instructure.com/courses/136715/modules>)



Lab 4: Interfaces (More) & Inheritance

Sensors and Data

1.1 Introduction

The purpose of this lab is to give you more practice with the concepts of interfaces, inheritance, typing and subclassing. We'll also continue to use polymorphism as you build out more intricate class hierarchies. This lab will also give you some experience integrating existing code, since we'll be building on some of the examples we developed in lecture.

1.2 What to do

One area of cyber-physical systems utilizes the **Internet of Things (IoT)** to collect data from a mesh of relatively inexpensive endpoints. This lab will use that idea by building on the Sensor example we worked on together during lecture. Since we've covered the topics of interfaces, sub-typing and subclassing, you have a bit of leeway with your design and implementation here, but you must pay particular attention to adhering to the contract and type specifications this lab asks for.



Based on our `sensorcomp` package from class, create a `WaterSensor`.

An `WaterSensor` is a concrete class that implements the `IDiscreteSensor` interface. In particular, a `WaterSensor` understands when its location is flooding, using this rule:

if the sensor's current reading (acquired by `takeNewReading()`) > 0.5 (representing inches for this sensor), then the sensor detects that the location is "**flooding**". If the current reading is ≤ 0.5 , the location is **NOT flooding**.
(NB: You will need to define a "conceptual mapping" for this status that adheres to the interface type specification)

You may use whatever OO design and implementation approach you deem suitable (composition, direct implementation, inheritance) but the **WaterSensor must adhere to the discrete sensor protocol/contract**. In case you want to develop "from scratch", here is the two interfaces we worked on during lecture:

```
package sensorcomp;

public interface ISensor {
    double takeNewReading();
    double lastReading();
}
```

```
package sensorcomp;

public interface IDiscreteSensor extends ISensor {
    boolean status();
    void flipStatus();
    void setStatus(boolean value);

    default double lastReading() { return 0;}

    @Override
    default double takeNewReading() {return 0;}
}
```

Our server tests will be providing synthetic test data via a **SensorData** class similar to what we developed in class. We've refactored that class to work with water data, so for your submission, use this version:

```
package sensorcomp;

public class SensorData {
    private static double [] readings = {0.1, 0.4, 0.0, 0.51, 0.5, 0.7, 0.0, 2.2, 1.0};
    private static int counter = 0;

    static public double currentReading() {
        int value = counter;
        counter++;
        if(counter >= readings.length) {
            counter = 0;
        }
    }
}
```

```
    }  
    return readings[value];  
}  
public static void reset() {  
    counter = 0;  
}  
}
```

If you develop your solution "from scratch" be sure to use `SensorData.currentReading()` as the source of your sensor's data stream, similar to what we did with our `AtmosphericSensor` class:

```
@Override  
public double takeNewReading() {  
    // Simulate a sensor reading  
    this.lastValue = this.currentValue; // save previous  
    this.currentValue = SensorData.currentReading();  
    return this.currentValue;  
}
```

Write an appropriate number of JUnit tests to validate your new sensor class. You should also perform a "smoke test" with the following code

```
import sensorcomp.ISensor;  
import sensorcomp.WaterSensor;  
import sensorcomp.IDiscreteSensor;  
import sensorcomp.SensorData;  
  
public class Main {  
  
    public static void main(String[] args) {  
        ISensor sensor = new WaterSensor();  
        for(int i=0; i< 10; i++) {  
            System.out.println("Water reading = " + sensor.takeNewReading() + " inches");  
            System.out.println("Our basement is flooding (True/False): " + ((IDiscreteSensor) sensor).status());  
        }  
    }  
}
```

Notes

The initial code we wrote in lecture is found here: [sensorcomp \(https://northeastern.instructure.com/courses/136715/files/19876850?wrap=1\)](https://northeastern.instructure.com/courses/136715/files/19876850?wrap=1). [↓ \(https://northeastern.instructure.com/courses/136715/files/19876850/download?download_frd=1\)](https://northeastern.instructure.com/courses/136715/files/19876850/download?download_frd=1) . Feel free to download and use this as your starting point, OR you may develop your solution directly from the "contracts" (interfaces) provided. Either way, ensure you conform to the two interfaces as specified: `ISensor` and `IDiscreteSensor`.

For each method you write:

- Design the signature of the method.
- Write Javadoc-style comments for that method.
- Write the body for the method.
- Write an appropriate number of JUnit tests to validate your solution. If you make use of inheritance, you can test the common superclass code ONCE and then reuse it in subclasses without rewriting new tests for that same code. If you override or extend superclass methods, you should write tests to validate the modifications you've introduced.

Feel free to create private "helper" methods if you need to do so.

Be sure to use access modifiers, private, default (no keyword), protected, and public appropriately.

Include JavaDoc for your classes and constructors as appropriate. You do not need to repeat JavaDoc already existing in a superclass or interface when you override a method. (This is true for the course in general.)

This lab is completely autograded, but you may ask one of our TAs to take a look at your solution if you want ideas for alternate approaches.

No UML class diagram is required for this lab, but we encourage you to update the one we developed in lecture to help guide your learning and understanding

Lab 4 - rubric

Full Credit - Pass	Partial Credit - Pass	No Credit
2 pts	1 pt	0 pt
Solution has clean compile, adheres to all interfaces/contracts,	Solution passes most automated tests.	Solution does not compile on server and/or passes few or no tests

passes all automated tests.		
-----------------------------	--	--