Inheritance and Polymorphism

COMP2603
Object Oriented Programming 1

Week 4

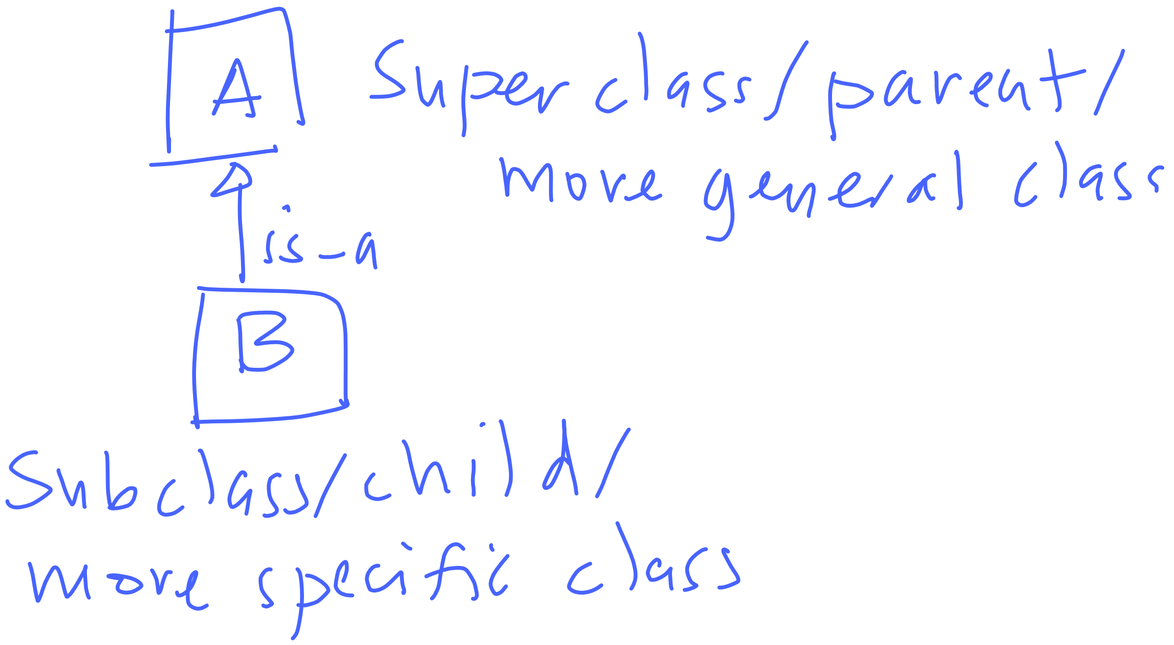
Outline

- Inheritance
  - Creating and Manipulating Subclass Instances
  - Constructors
  - Method Refinement ✓
  - Method Replacement ←
- Object Class
  - toString( )

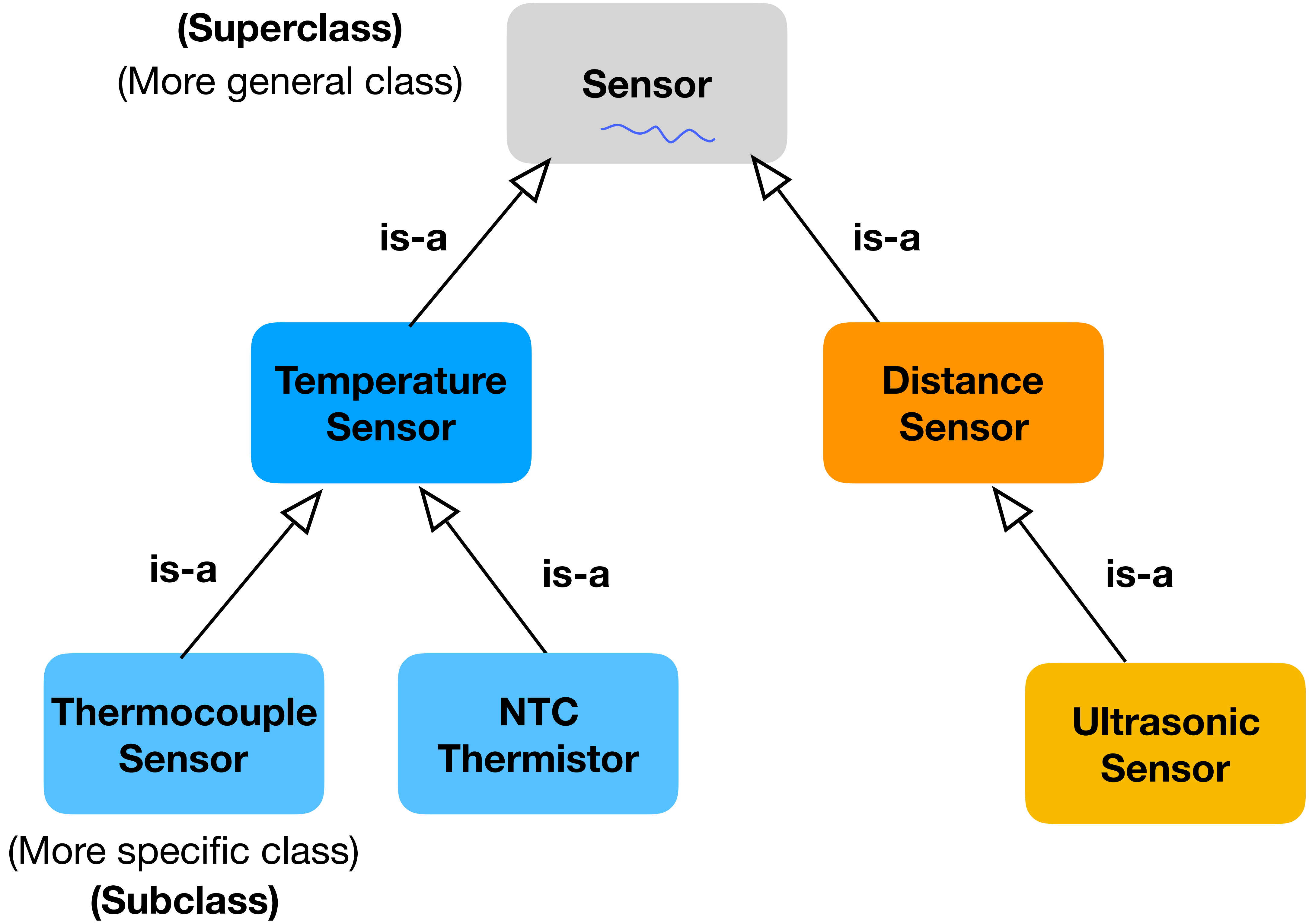
Reading: Chapter 9 - Inheritance and Polymorphism: P. Mohan 2013

Generalisation vs Specialisation

Generalisation is used to model a relationship between classes in which one class represents a more general concept and another class represents a more specialised concept.



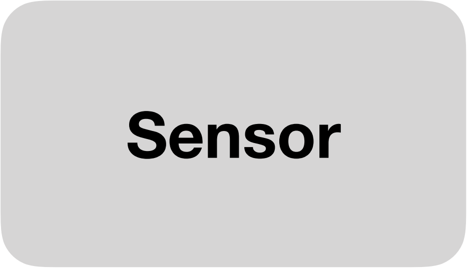
Example - Inheritance



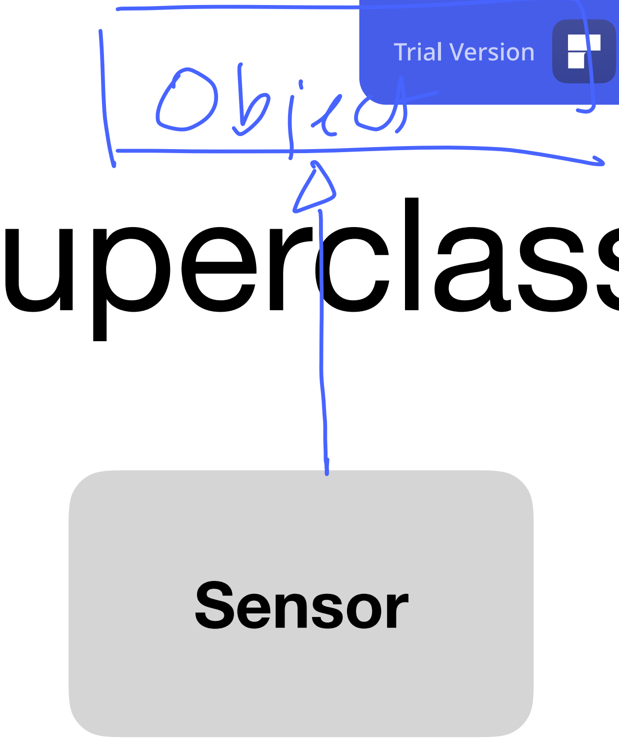
Example

```
public class Sensor{
  private String unit;
  private double currentValue;

  public Sensor(){ // constructor
  }
  public void setUnit(String u){
       unit = u;
  }
  public double getCurrentValue(){
       return currentValue;
  }
}
```

toString()


Sensor

Example: Instance of a Superclass



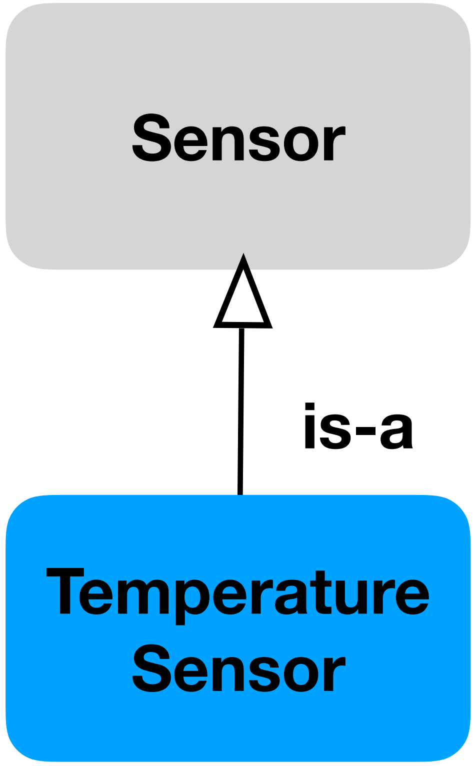
```
1. Sensor s = new Sensor(); // new Sensor object
2. s.setUnit(“centimetre”); // invoke a Sensor method
3. double v = s.getCurrentValue(); // invoke another Sensor method
4  String t = s.toString(); //invoke inherited method from Object
```

Sensor@19478C

All Java classes are subclasses of the Object class, therefore the methods of the Object class can be used on instances of any Java class.

Example Implementing Inheritance

```
public class TemperatureSensor extends Sensor{
  private double maxValue;
  private double responsivenessLapse;

  public void updateMaxValue(){..}

  public double getResponsivenessLapse(){..}
}
```



2

x

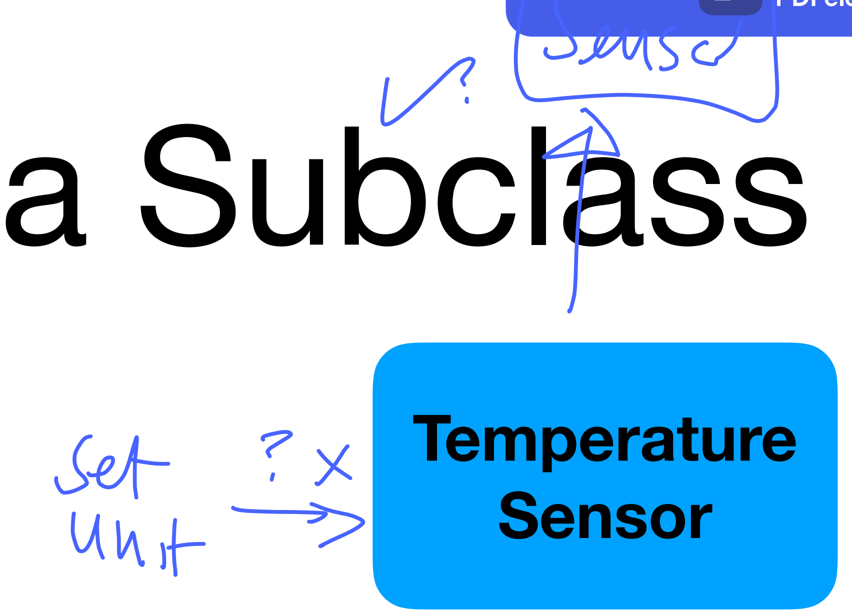
+2

=4

The TemperatureSensor class adds on two attributes, two new methods that are specialised to the TemperatureSensor.

It can still use the inherited public methods from its superclass: Sensor.

Example: Instance of a Subclass



Temperature Sensor

setunit

```
1 TemperatureSensor ts = new TemperatureSensor();
2 ts.updateMaxValue(); //invoke TemperatureSensor method
3 double l = ts.getResponsivenessLapse(); //invoke TemperatureSensor
method

4 ts.setUnit("centimetre"); // invoke a Sensor class inherited method
5. double v = ts.getCurrentValue(); // invoke another Sensor method
```

AC ×

→ #currentValue ×

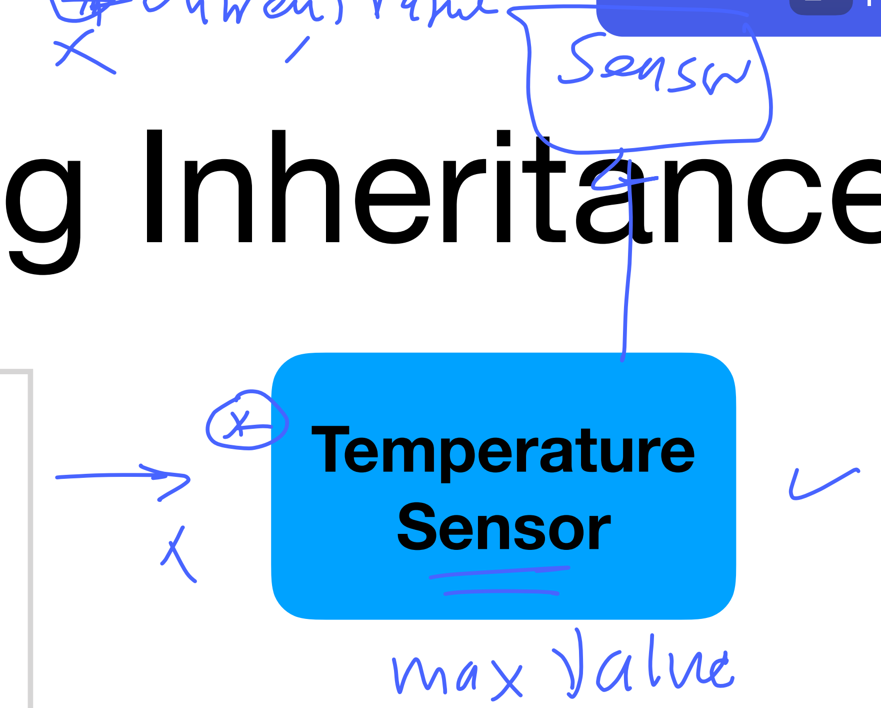
Example Implementing Inheritance

S TS

1)

2.

```
public void updateMaxValue(){
    if(currentValue > maxValue)
        maxValue = currentValue;
}
```



Temperature Sensor

This method generates a compilation error: currentValue is not defined in the TemperatureSensor class. Even though it is inherited, it cannot be accessed in this way because it is private.

Inherited Attributes and Methods

The access modifiers (protected, public) allow subclasses to use the inherited methods and the attributes from the superclass.

```
public void updateMaxValue(){
    if(getCurrentValue() > maxValue)
       maxValue = getCurrentValue();
}
```

Solution #1: use the accessors and mutators

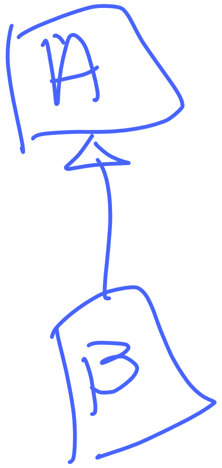
```
public class Sensor{
  private String unit;
  protected double currentValue;
  //… rest of class
}
```

Solution #2: change the access modifier in the superclass to protected.

Constructors

The constructors in the examples have been simple, no-argument constructors. Subclasses therefore have been using no-argument constructors (provided by default).

When arguments are required by a superclass constructor, the subclasses must supply these ① parameters and explicitly invoke the parent constructor. ②

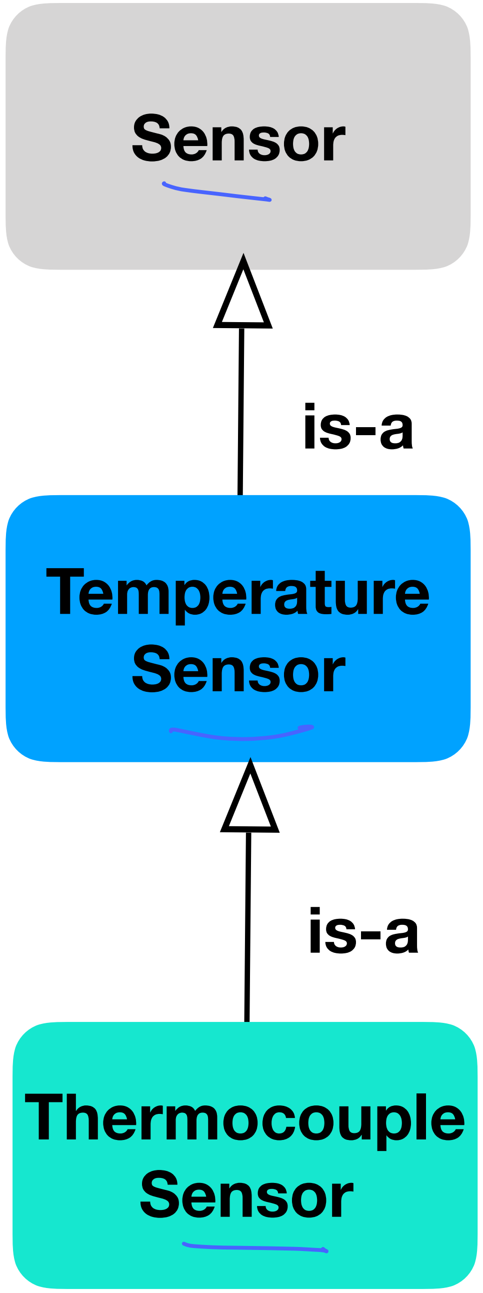


```
public class A {
  public A(String s) {
  }
}

public class B {
  public B() {
  }
}
```

Example Implementing Inheritance

```
public class TemperatureSensor extends Sensor{
    //state variables as in previous slides
    // simple no-argument constructor
1   public TemperatureSensor(){
    }
    //overloaded constructor
2   public TemperatureSensor(double lapse){
      responsivenessLapse = lapse;
     }
}

public class ThermocoupleSensor extends TemperatureSensor{
1    public ThermocoupleSensor(double tclapse){
        super(tclapse); // invoke the direct parent constructor
     }
}
```

this



Sensor

is-a

Temperature Sensor

is-a

Thermocouple Sensor

Example Implementing Inheritance

```
1. Sensor s = new Sensor();
2. s.getCurrentValue();
```

✓ default no-argument ()

✓

```
3. TemperatureSensor ts = new TemperatureSensor();
4. ts.getCurrentValue();
```

① no-argument
② overloaded cons)-
(double)

✓

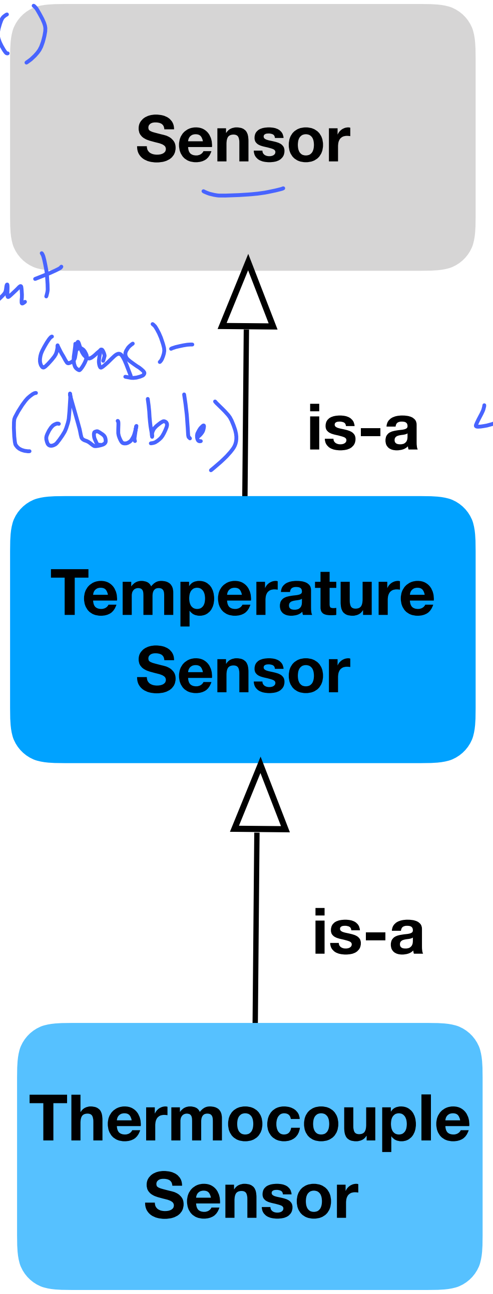
```
5. TemperatureSensor ts2 = new TemperatureSensor(5);
6. ts2.getCurrentValue();
```

✓

```
7. ThermocoupleSensor tcs = new ThermocoupleSensor(100);
8. tcs.getCurrentValue();
```

✓

9. tcs = new ThermocoupleSensor();



getCurrentValue()
public

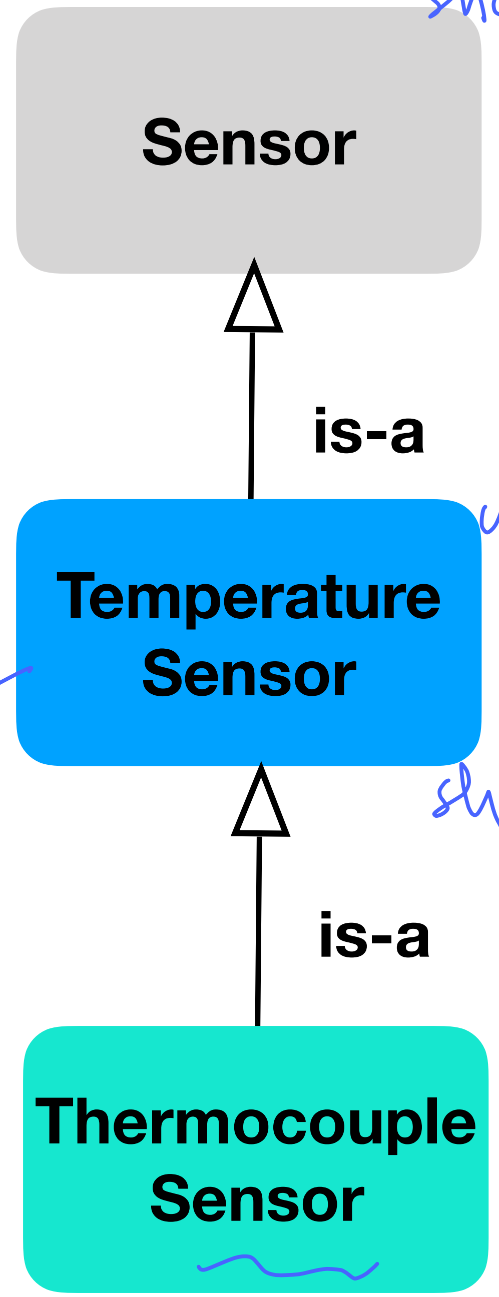
✓

direct subclass of Sensor

indirect subclass of Sensor.

Example Implementing Inheritance

```
1. Sensor s = new Sensor();
2. s.getCurrentValue();

3. TemperatureSensor ts = new TemperatureSensor();
4. ts.getCurrentValue();
5. ts.updateMaxValue();

6
ThermocoupleSensor tcs = new ThermocoupleSensor(100);
7. tcs.getCurrentValue();
8 ~~ts~~.updateMaxValue();
  tcs
```



slides

update Max Value()

slide 7

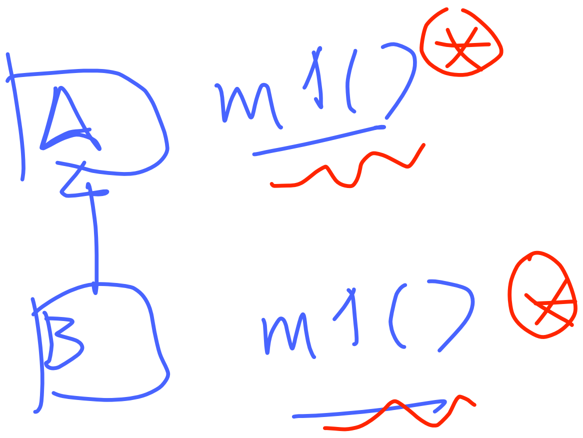
Method Refinement and Replacement

A subclass can change the methods inherited from its superclass. The method signature has to be maintained for this to happen.

- Method Refinement
- Method Replacement

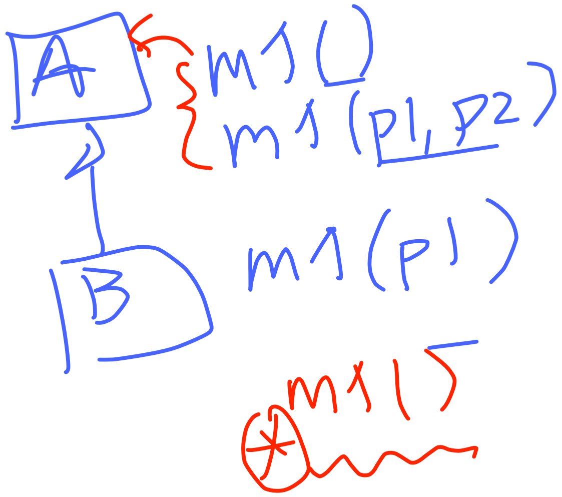
overriding

same name
same params



overload

same name
diff params



1

Method Replacement

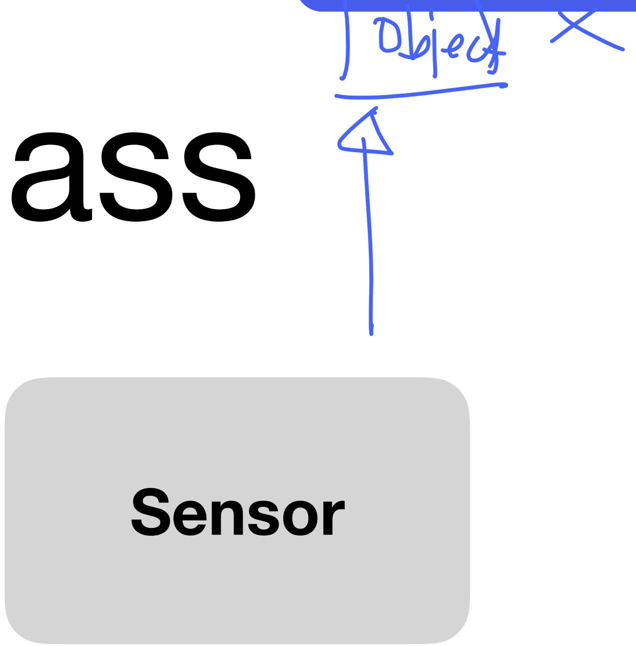
- Method Replacement: the method in the subclass completely replaces the inherited behaviour. The superclass code is never executed. New behaviour is introduced
- This is essentially overriding.

2

Method Refinement

- Method Refinement: the method in the subclass adds some extra behaviour of its own while maintaining the original behaviour that was inherited. The superclass code is executed with some extra code in the subclass.
  - The keyword `super` is used to invoke the superclass' method within the refined method in the subclass.

Method in Parent Class



```
public class Sensor{

  public String toString(){
      String details = “”;
      details = “Current reading: “ +currentValue + units;
      return details;
    }
}
```

method
replacement ✓

Suppose we write a toString( ) method in the Sensor class.
All of the subclasses will inherit this method.

Example: Direct Instances

```
Sensor s = new Sensor(15, “C”); // assume valid constructor
System.out.println(s.toString());

Sensor s2 = new Sensor(30, “F”); // assume valid constructor
System.out.println(s2.toString());
```

Output:

Current Reading: 15C
Current Reading: 30F

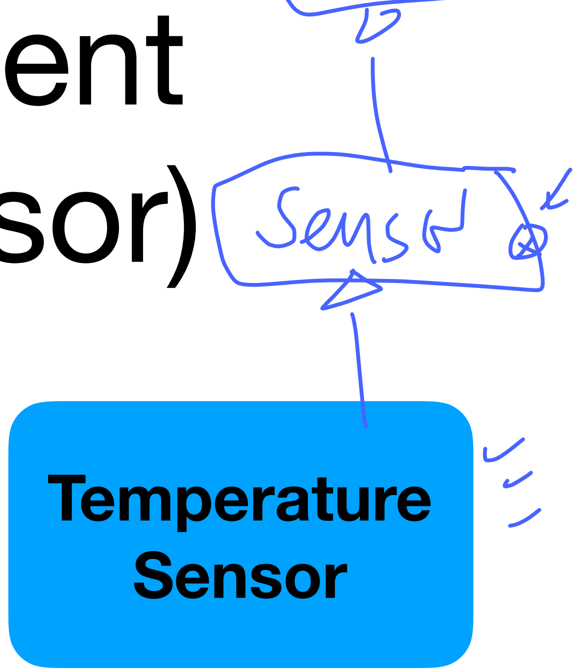
Example: Subclass Instances

```
/* assume valid constructor in subclass that invokes the
appropriate superclass constructor*/
TemperatureSensor ts = new TemperatureSensor(50, “F”);
System.out.println(ts.toString());

TemperatureSensor ts2 = new TemperatureSensor(70, “F”);
System.out.println(ts2.toString());
```

Output:

Current Reading: 50F
Current Reading: 70F

Example - Method Refinement (Inherited Method from Sensor)



Temperature Sensor

```
public class TemperatureSensor{

1.  public String toString(){
2.      String sensorDetails = super.toString( );
3,      sensorDetails +=  “ Max temp: “ + maxValue;
4,      return sensorDetails;
    }
}
```

Suppose we write a toString( ) method in the TemperatureSensor class that uses the inherited method and adds some extra functionality class.

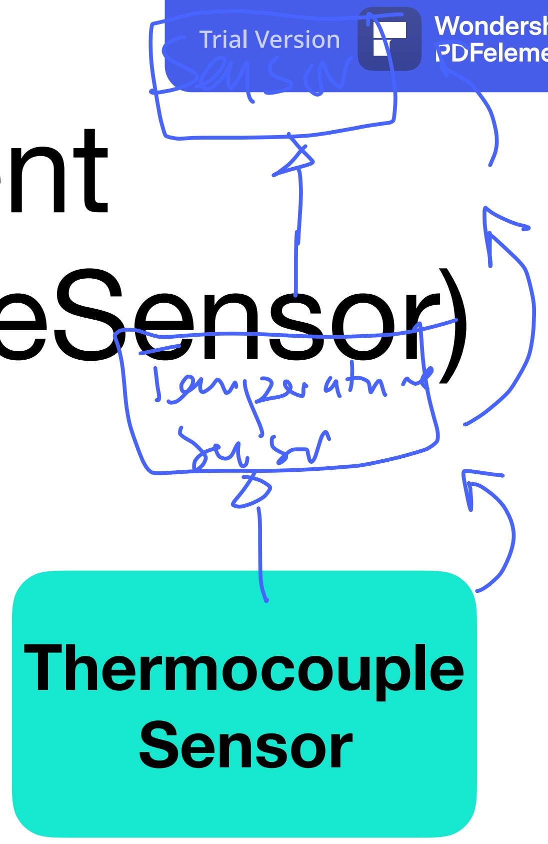
Example: Instances (Method Refinement)

```
1. Sensor s = new Sensor(15, “C”); // assume valid constructor
2. System.out.println(s.toString());

3. TemperatureSensor ts = new TemperatureSensor(50, “F”);
4  System.out.println(ts.toString()); // refines / reuses

5. TemperatureSensor ts2 = new TemperatureSensor(70, “F”);
6. System.out.println(ts2.toString());
```

Output:

```
1,2  Current Reading: 15C
                                 +
3,4  Current Reading: 50F Max temp: 50F
5,6  Current Reading: 70F Max temp: 70F
```

Example - Method Refinement (Inherited Method from TemperatureSensor)



extends TemperatureSensor.

```
public class ThermocoupleSensor{
  private double mv; // millivoltage

  public void convertMV(){…} // converts MV to C

  public String toString(){
      convertMV();
      String sensorDetails = super.toString();
      sensorDetails += " Thermocouple: " + mv;
      return sensorDetails;
   }
}
```

refinement

Suppose we write a toString( ) method in the ThermocoupleSensor class that also uses the inherited method.

Example: Instances (Method Refinement)

```
/* assume valid constructor in subclass that invokes the
appropriate superclass constructor*/
                                                        mV
1. ThermocoupleSensor tcs = new ThermocoupleSensor(3.41);
//assume 3.41mv converts to 100C
2. System.out.println(tcs.toString());
```

Output:

Current Reading: 100C   Max temp: 100C   Thermocouple: 3.41

gp.   p.   child,

Example: Instances (Method Refinement)

```
1. Sensor s = new Sensor(15, “C”); // assume valid constructor
   System.out.println(s.toString());


2, TemperatureSensor ts = new TemperatureSensor(50, “F”);
   System.out.println(ts.toString());


3, ThermocoupleSensor tcs = new ThermocoupleSensor(3.41);
   System.out.println(tcs.toString());
```

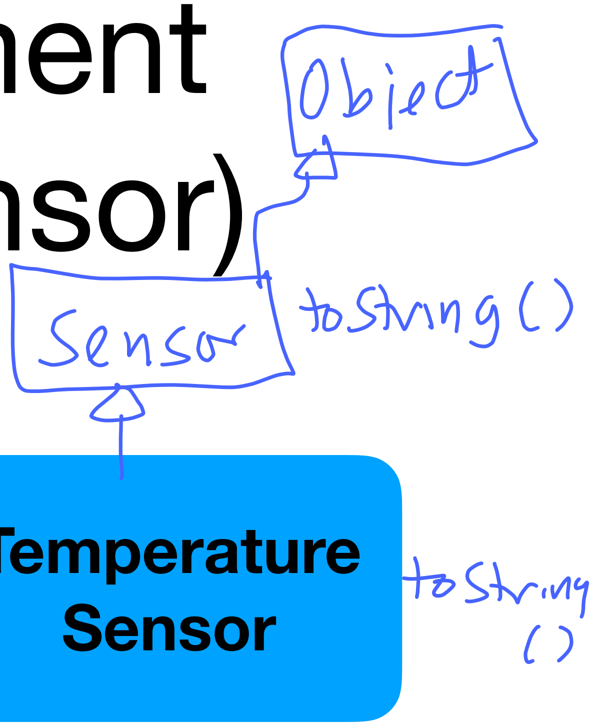
Output:

```
1. Current Reading: 15C
2. Current Reading: 50F  Max temp: 50F
3. Current Reading: 100C    Max temp: 100C    Thermocouple: 3.41
```

Example - Method Replacement (of Inherited Method from Sensor)

Super.method()

refinement



/*assuming protected variables in the Sensor class*/

extends Sensor

```
public class TemperatureSensor{

1. public String toString(){
2.     String sensorDetails = currentValue + “ “ + units;
3.     return sensorDetails;
     }
}
```

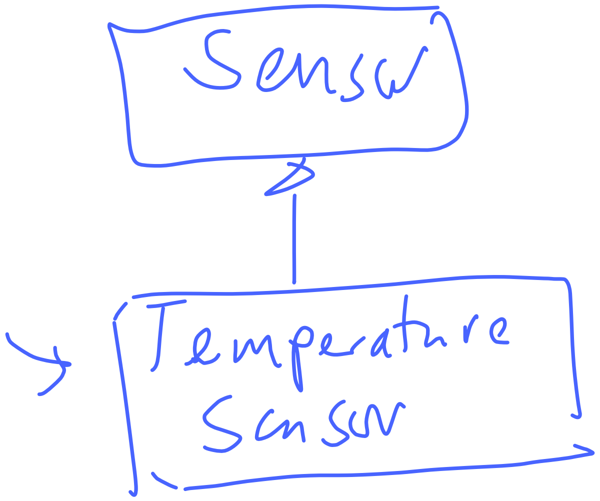
Here we write a toString( ) method in the TemperatureSensor class completely replacing the inherited toString( );

Example: Instances (Method Replacement)

```
1. Sensor s = new Sensor(15, “C”); // assume valid constructor
2. System.out.println(s.toString());

3. TemperatureSensor ts = new TemperatureSensor(50, “F”);
4. System.out.println(ts.toString()); // replaced

5. TemperatureSensor ts2 = new TemperatureSensor(70, “F”);
6. System.out.println(ts2.toString());
```

Output:

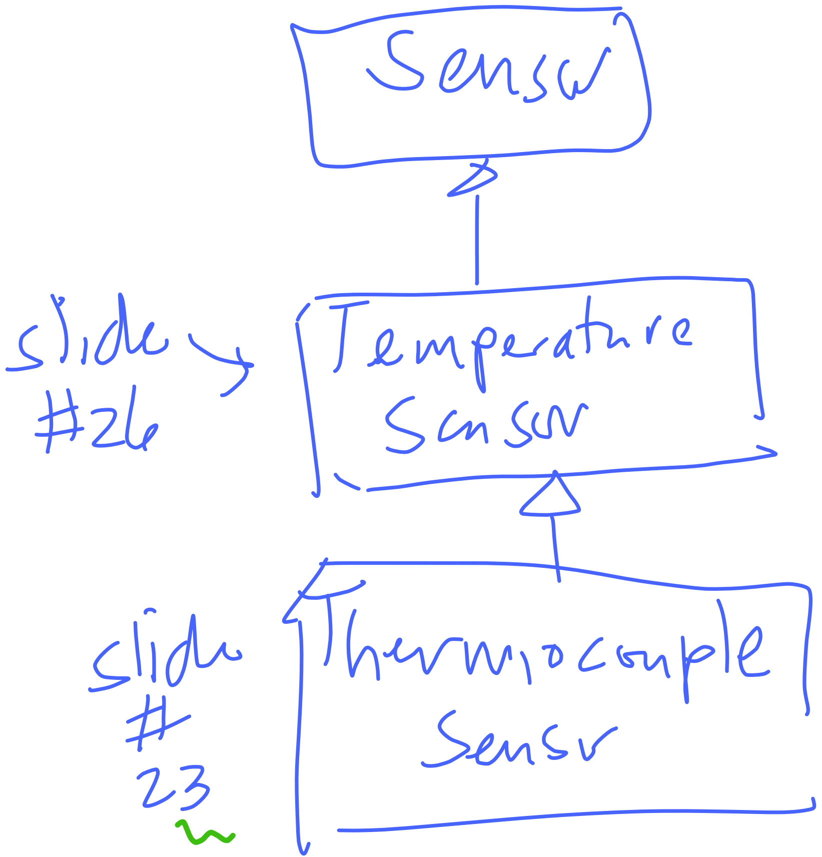
```
Current Reading: 15C
50F
70F
```



Example: Instances (Method Replacement)

polymorphism

```
1 Sensor s = new Sensor(15, “C”); // assume valid constructor
2 System.out.println(s.toString());

3 TemperatureSensor ts = new TemperatureSensor(50, “F”);
4 System.out.println(ts.toString());

5 ThermocoupleSensor tcs = new ThermocoupleSensor(3.41);
6 System.out.println(tcs.toString());
```



Output:

Current Reading: 15C

50F

100C Thermocouple: 3.41

Temp Sensor toString

Thermocouple to string (refinement)

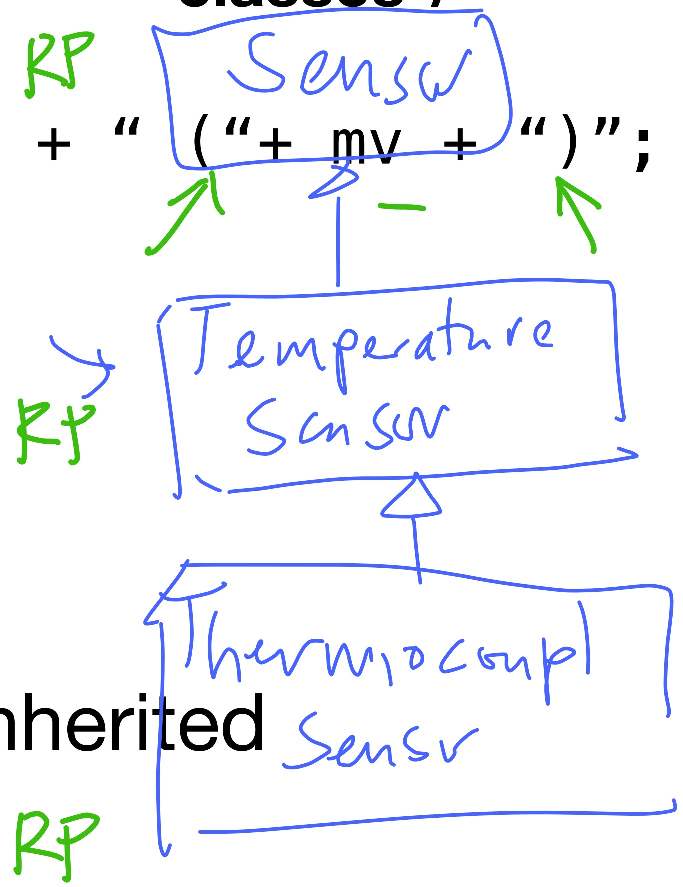
Example - Method Replacement (of Inherited Method from TemperatureSensor)

Thermocouple Sensor

/*assuming protected variables in the Sensor and TemperatureSensor classes*/

```
public class ThermoCoupleSensor{

  public String toString(){
      String sensorDetails = currentValue + units + “ (“+ mv + “)”;
      return sensorDetails;
   }
}
```

700 F



Another example of completely replacing the inherited toString( ) inherited from the parent class.

Example: Instances (Method Replacement)

1

```
Sensor s = new Sensor(15, “C”); // assume valid constructor
System.out.println(s.toString());
```

2

```
TemperatureSensor ts = new TemperatureSensor(50, “F”);
System.out.println(ts.toString());
```

3

```
ThermocoupleSensor tcs = new ThermocoupleSensor(3.41);
System.out.println(tcs.toString());
```

Output:

1 **Current Reading: 15C**

2. **50F**

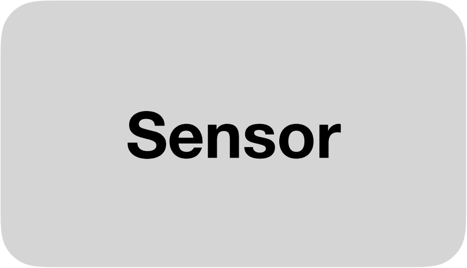
3, ~~**100C Thermocouple: 3.41**~~

700 F (3.41)

Example - Method Replacement (Inherited Method from Object)

extends Object

```
public class Sensor{

  public String toString(){
      String details = “”;
      details = “Current reading: “ +currentValue + “ “+ units;
      return details;
   }
}
```


Sensor

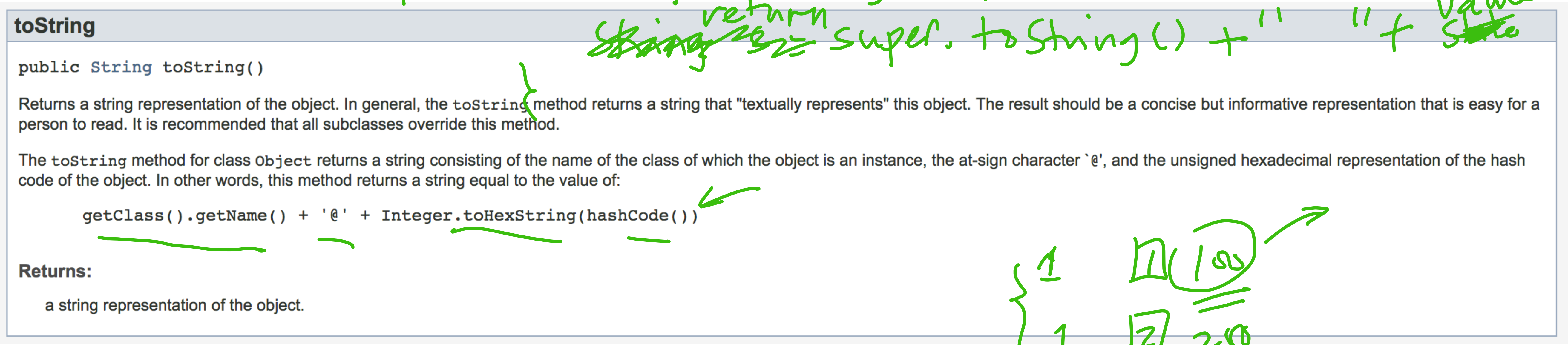
This is also an example of method replacement since the toString( ) method was inherited from the Object class in the first place.

Object Class: toString( )

public String toString() {

~~String~~ return ~~s =~~ super.toString() + " " + current value ~~state~~

}



toString

```
public String toString()
```

Returns a string representation of the object. In general, the `toString` method returns a string that "textually represents" this object. The result should be a concise but informative representation that is easy for a person to read. It is recommended that all subclasses override this method.

The `toString` method for class `Object` returns a string consisting of the name of the class of which the object is an instance, the at-sign character `` `@' ``, and the unsigned hexadecimal representation of the hash code of the object. In other words, this method returns a string equal to the value of:

```
getClass().getName() + '@' + Integer.toHexString(hashCode())
```

Returns:

a string representation of the object.

{ 1 [1] 100

1 [2] 200

[3] 300

Sensor@4A123B 35

This means that every object has a toString( ) method and we are encouraged to override this method.

String API: https://docs.oracle.com/javase/7/docs/api/java/lang/Object.html#toString()

Exercises

HW

What happens when we remove the various toString( ) methods that we wrote from the classes in different permutations?

Slide #: toString( ) method code
RP: Method replacement
RF: Method refinement

Consider the instances on Slide 28, and state how the output changes when the following toString( ) methods are included in the various classes:

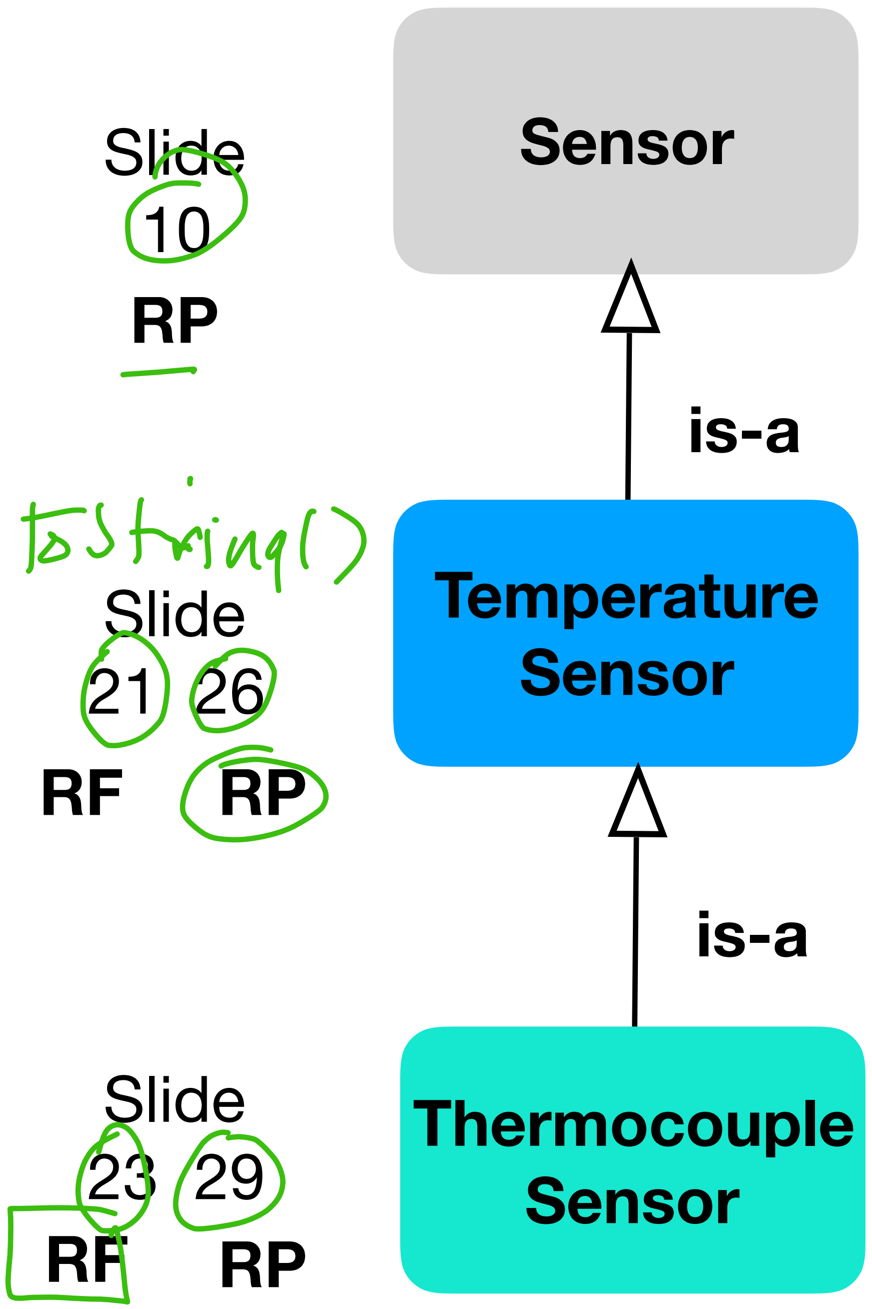
1) Slide 10, 21, 23

2) Slide 10, 26, 29

3) Slide 21, 23

4) Slide 26

5) Slide 29



Slide 10
RP

toString()
Slide 21 26
RF RP

Slide 23 29
RF RP

Reading

Reading: Chapter 9 - Inheritance and Polymorphism: P. Mohan 2013