Relationships between Objects

Dependencies, Associations and Generalisations

COMP2603
Object Oriented Programming 1

Week 3 Lecture 1

Outline

- Types of Relationships in Object-Oriented Programming
  - Dependencies
  - Associations
  - Generalisations
- Implementing Relationships
  - Dependencies
  - Associations
  - Generalisations

Relationships

Classes, like objects, do not exist in isolation. Very often, an object-oriented program consists of a set of interacting objects whose classes are related in some way.

Relationships between classes are established to either:

- Indicate some sort of sharing between the classes
- Indicate a semantic connection between the classes

Example

Sensor

Telemetry Data

Temperature Sensor

Distance Sensor

Air Conditioner

Thermocouple Sensor

Ultrasonic Sensor

Room

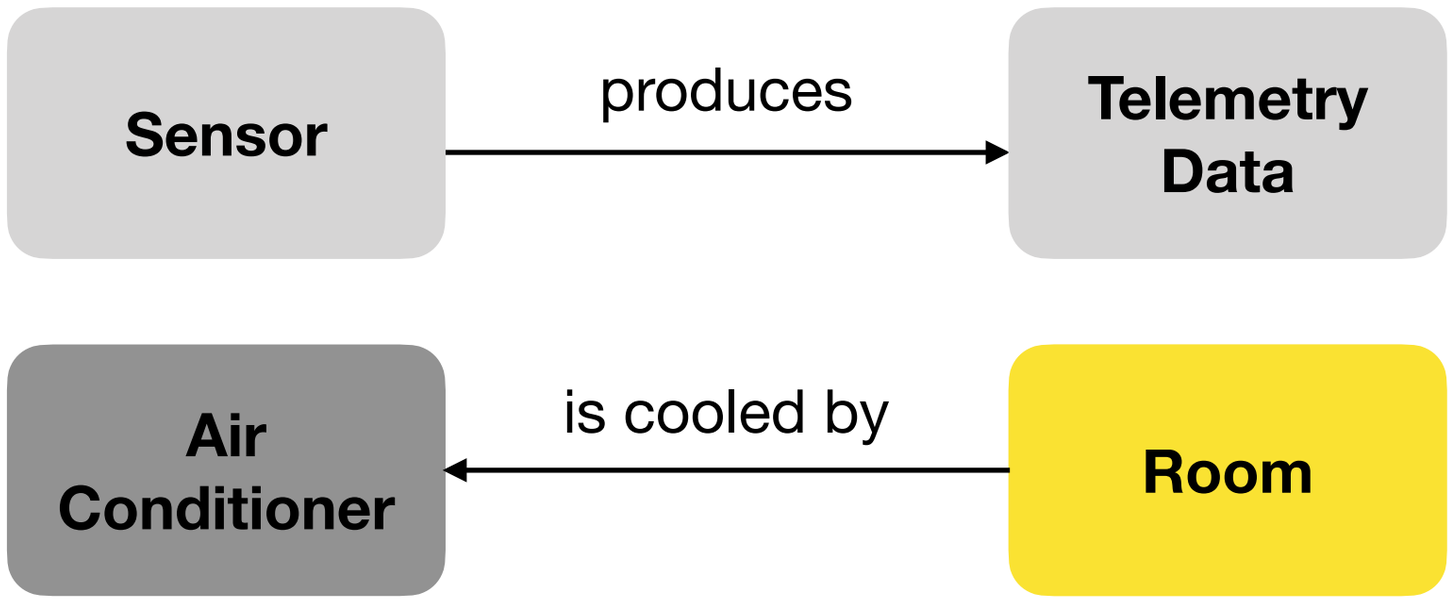
Kinds of Relationships

There are three basic kinds of relationships:

1. Association/Dependency (uses)
2. Generalisation/Specialisation (is-a)
3. Composition/Aggregation (part-of)

Associations

An association denotes a semantic dependency between objects. The direction of this association can be bidirectional or can be navigated specifically in one direction.



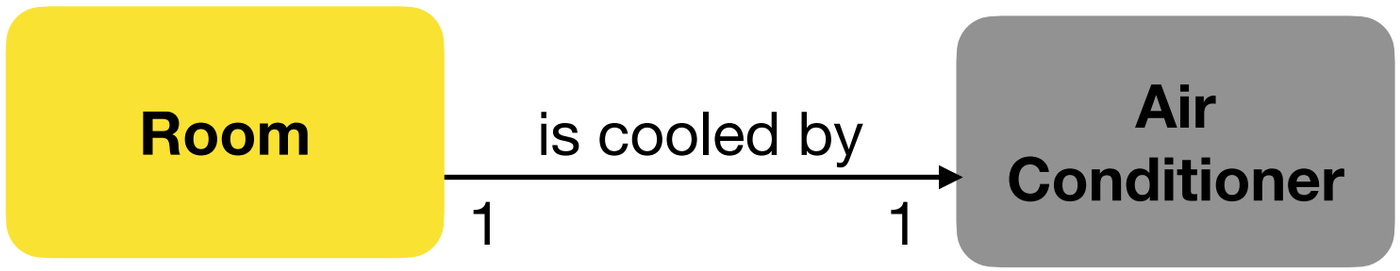
Cardinality

The cardinality of an association specifies the number of participants in the semantic relationship.

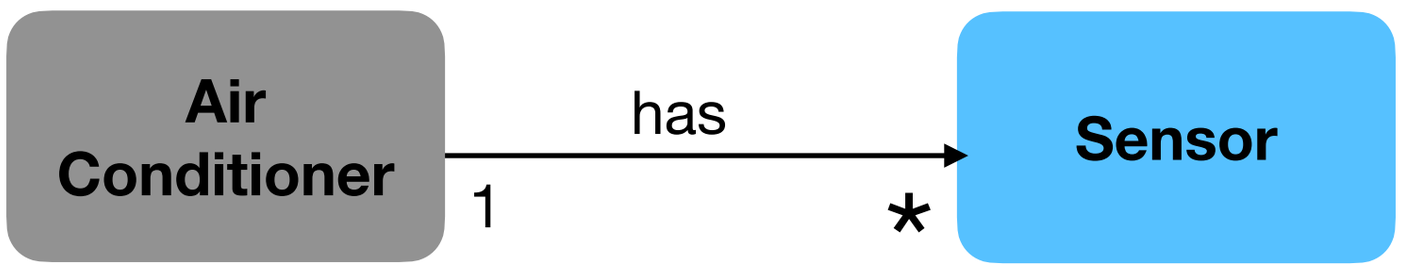
Three common types of cardinality are:

- One-to-one (narrow)
- One-to-many
- Many-to-many

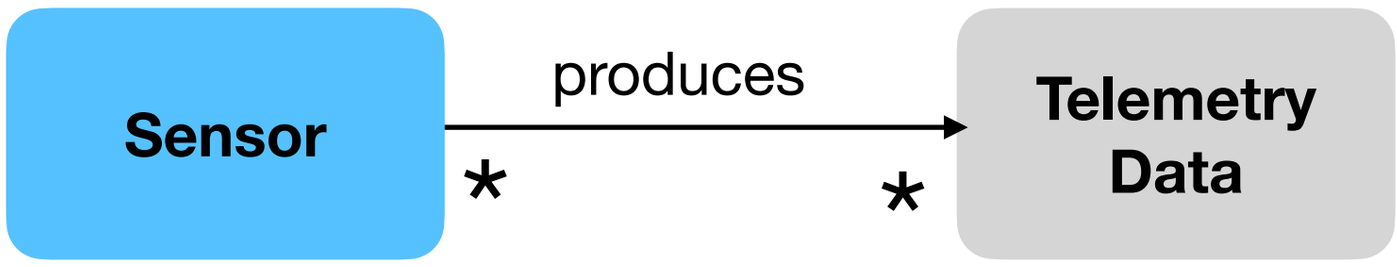
Example- Association



one-to-one association



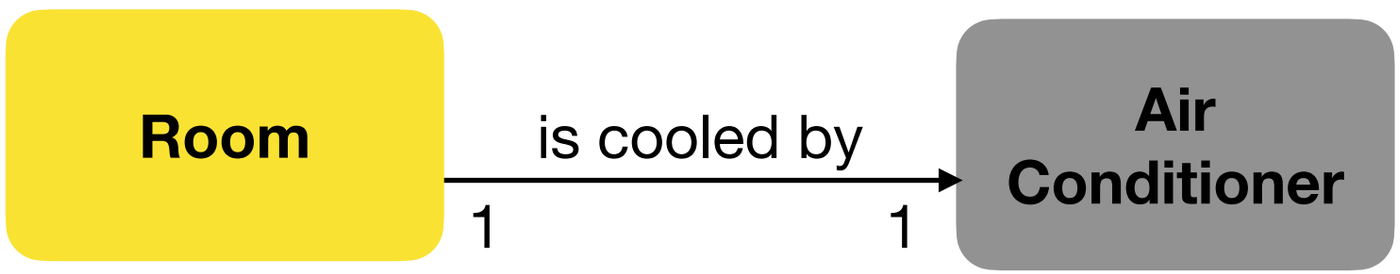
one-to-many association



many-to-many association

Example - Implementing Associations

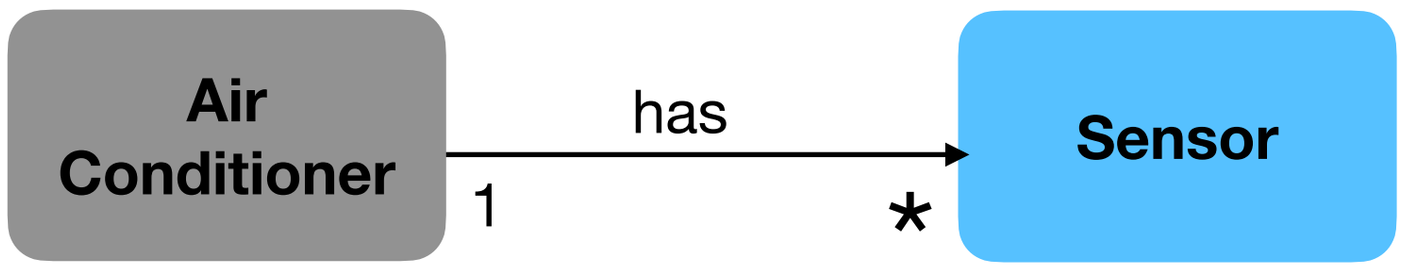
```
public class Room{
    private AirConditioner ac;
    public Room( ){
      ac = new AirConditioner( );
    }
}
```



one-to-one association

Example - Implementing Associations

```
public class AirConditioner{
    private Sensor[] sensors;
    public AirConditioner( ){
     sensors = new Sensor[3];
    }
}
```



one-to-many association

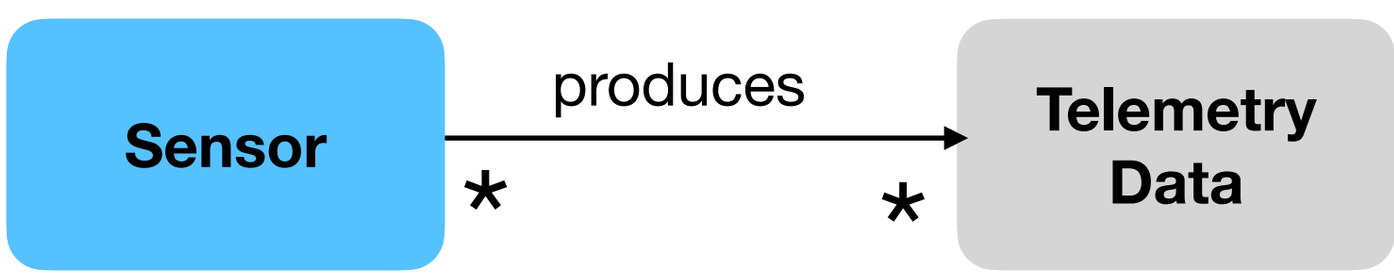
Example - Implementing Associations

```
public class Sensor{
      private TelemetryData[] data;
      private int currIndex;
      public Sensor( ){
       data = new TelemetryData[1000];
       currIndex = 0;
      }
      public TelemetryData readValue(Date date){
          for(int i = 0; i < currIndex; i++)
              if(data[i].getDate( ).equals(date))
                  return data[i];
          return null;
      }

}
```

```
public class TelemetryData{
      private Date date;
      private double value;
      public TelemetryData(double value){
       date = new Date();
       this.value = value;
      }
      public double getValue( ){
          return value;
      }
      public Date getDate( ){
          return date;
      }

  }
```



many-to-many association

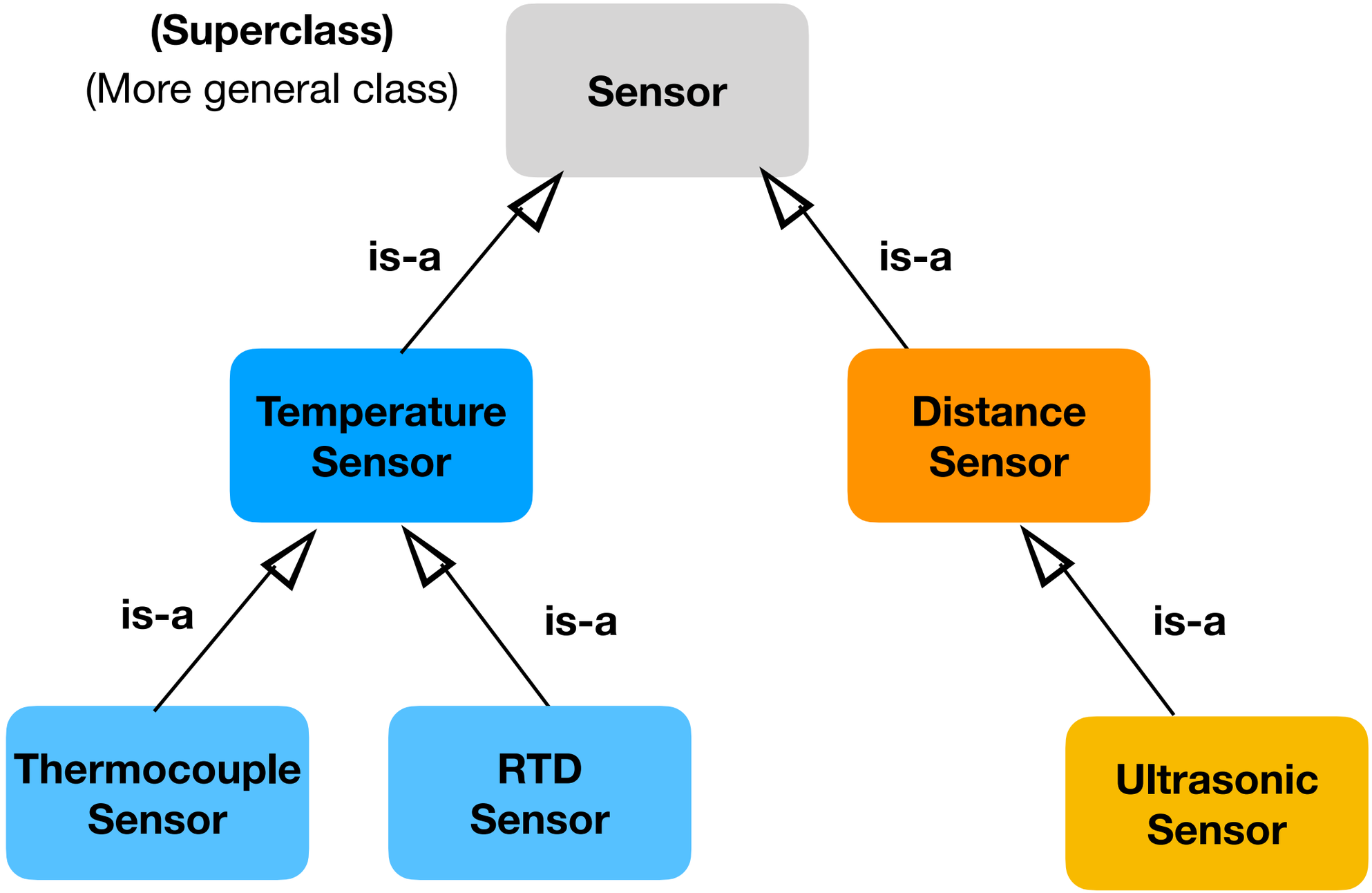
Inheritance

Inheritance is a relationship among classes wherein one class shares the **structure** and/or **behaviour** defined in one (single inheritance) or more (multiple inheritance) other classes (Booch 1994).

A class from which another class inherits its structure and/or behaviour is called the **superclass**. A class that inherits from one or more classes is called a **subclass**

Example - Inheritance

(Superclass)
(More general class)



(More specific class)
(Subclass)

Example -Implementing Inheritance

```
public class Sensor{
    private String unit;
    private double maxRangeValue;
    private double minRangeValue;
    private int responsivenessLapse;

  public Sensor(){
  }
  //accessors & mutators
}
```

Sensor

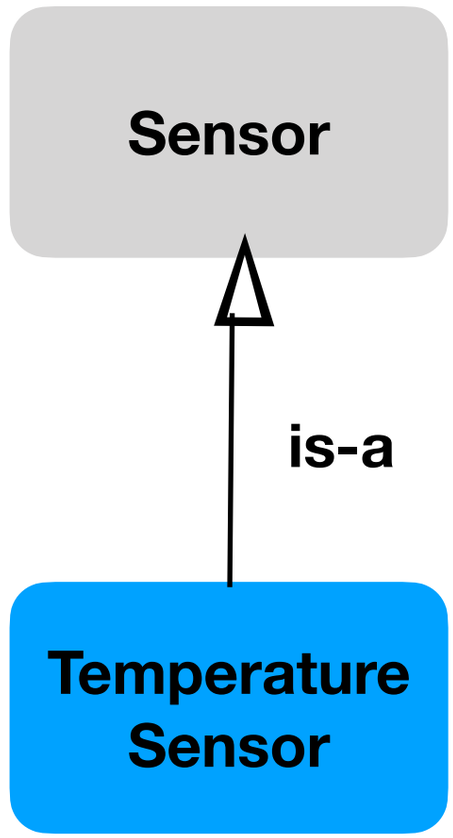
```
Sensor s = new Sensor();
s.setUnit();
s.getMinRangeValue();
s.getUnit();
```

Client class

Example Implementing Inheritance

```
public class Sensor{
   private String unit;
   private double maxRangeValue;
   private double minRangeValue;
   private int responsivenessLapse;

   public Sensor(){
   }
   //accessors & mutators
}
```




```
public class TemperatureSensor extends Sensor{
}
```

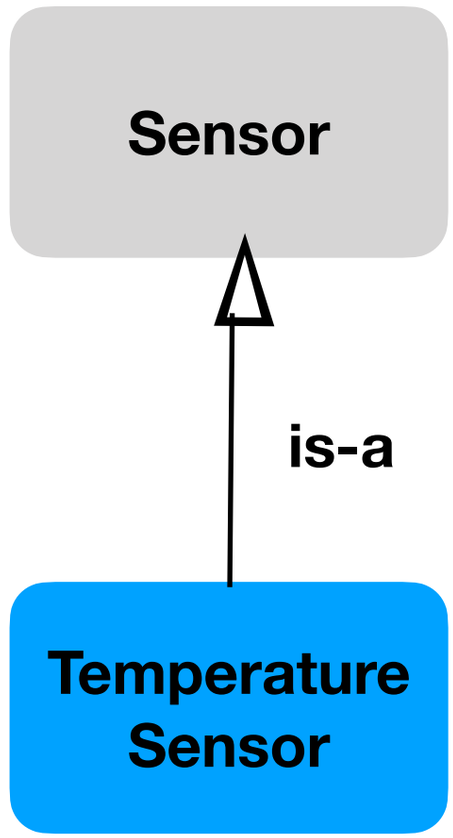
```
TemperatureSensor t = new TemperatureSensor();
t.setUnit();
t.getMinRangeValue();
t.getUnit();
t.unit = “Celsius”;
```

Client class

Example Implementing Inheritance

```
public class Sensor{
   protected String unit;
   protected double maxRangeValue;
   protected double minRangeValue;
   protected int responsivenessLapse;

   public Sensor(){
   }
   //accessors & mutators
}
```



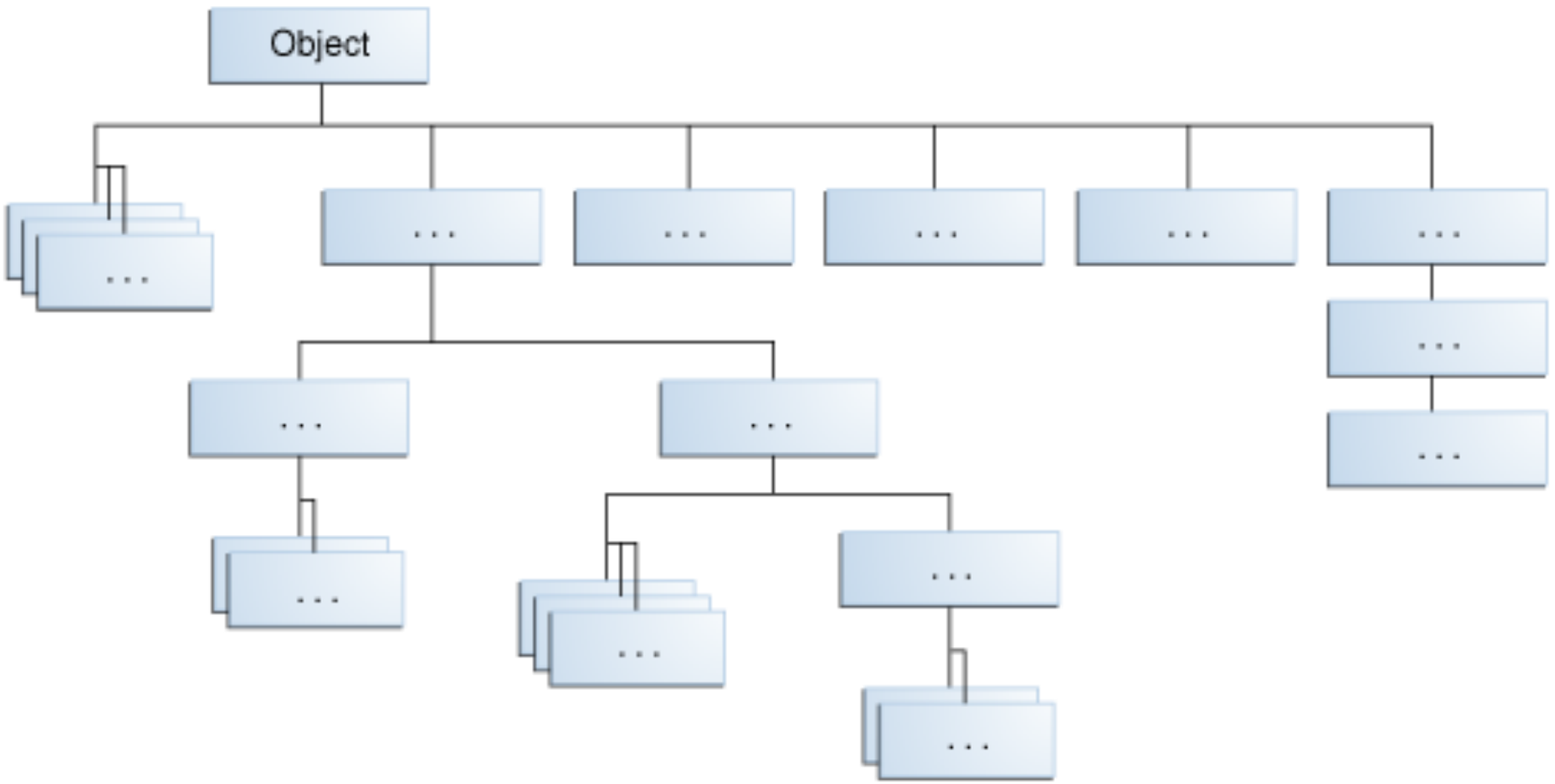

```
public class TemperatureSensor extends Sensor{
}
```

```
TemperatureSensor t = new TemperatureSensor();
t.setUnit();
t.getMinRangeValue();
t.getUnit();
t.unit = “Celsius”;
```

Client class

Object Class

The `Object` class, defined in the **`java.lang`** package, defines and implements behavior common to all classes—including the ones that you write. In the Java platform, many classes derive directly from **`Object`**, other classes derive from some of those classes, and so on, forming a hierarchy of classes.



Object Class

Some of the methods inherited from Object are:

- `public boolean equals(Object obj)`
  Indicates whether some other object is "equal to" this one.
- `public int hashCode()`
  Returns a hash code value for the object.
- `public String toString()`
  Returns a string representation of the object.

String Class

The String class is an immutable class. It has no mutators and it is impossible to change the state of the class after it has been created.

Several methods are available for use when you create a String object.

```
String s = “chickens”;
boolean plural = s.endsWith(“s”);
```

https://docs.oracle.com/javase/7/docs/api/java/lang/String.html

References

- Booch, Grady. (1988) OBJECT-ORIENTED ANALYSIS AND DESIGN
- Mohan, Permanand (2013) FUNDAMENTALS OF OBJECT-ORIENTED PROGRAMMING IN JAVA