# Polymorphism

COMP2603
Object Oriented Programming 1

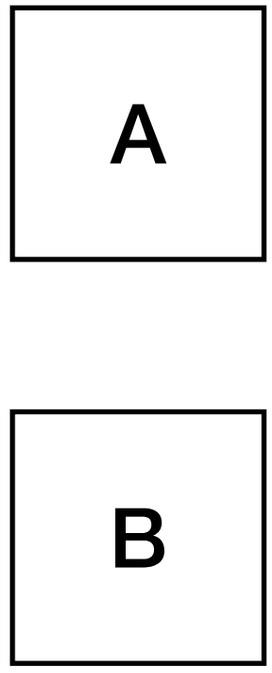
Week 5

# Outline

- Principle of Substitutability
- Polymorphism
- Method Binding
  - Static vs Dynamic Object Typing
  - Reverse Polymorphism
    - Casting & instanceof
- Object Equality

# Principle of Substitutability (PoS)

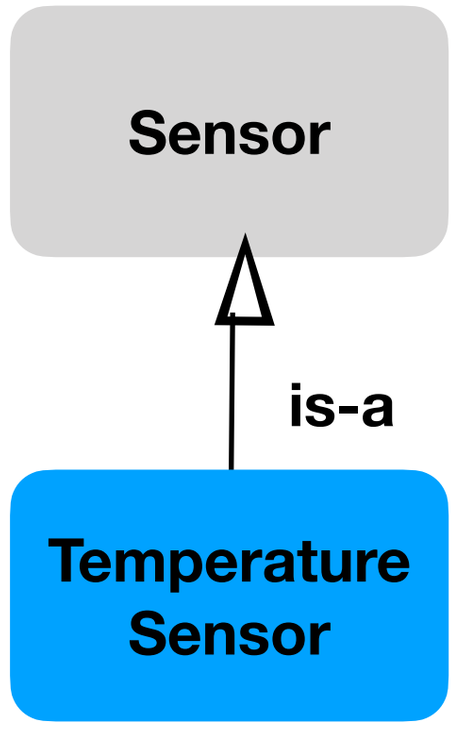
If we have two classes **A** and **B**, such that **B** is a subclass of **A** (even indirectly), it should be possible to substitute instances of class **B** for instances of class **A** in any situation with no observable effect.



# Principle of Substitutability (PoS)

Suppose we declare an instance of a Sensor class

```
Sensor s;
```



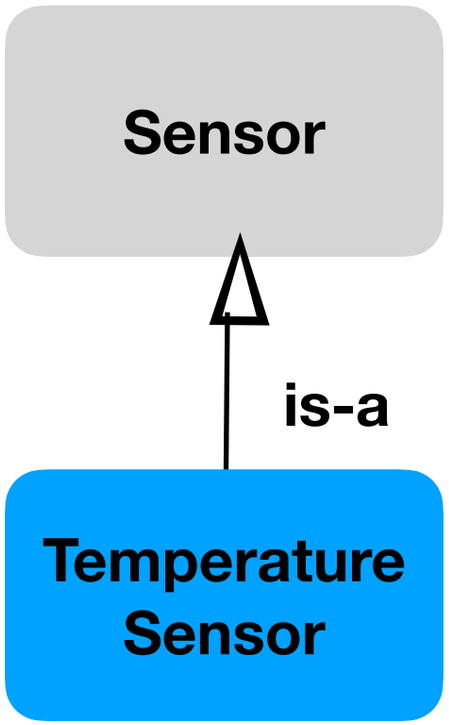

We can assign an instance of the TemperatureSensor class to the Sensor instance based on the Principle of Substitutability.

```
s = new TemperatureSensor();
```

It is safe to do this because TemperatureSensor has all of the behaviour and state owned by Sensor. Therefore we can invoke and use these features without any problems.

# Example - PoS

```
Sensor s1;                                  ✓
TemperatureSensor ts1;                      ✓
s1 = new Sensor();                          ✓
ts1 = new TemperatureSensor();              ✓
s1 = ts1; // works due to PoS               ✓
```

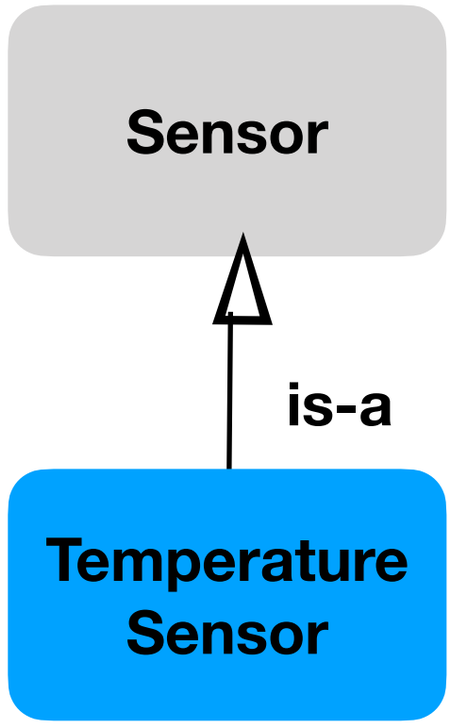


# Subtype vs Subclass

**Subtype** refers to a specialisation relationship in which the Principle of Substitutability is maintained.

This is distinguished from the **subclass** relationship, which may or may not satisfy this principle.

# Example - PoS




We cannot assign a superclass to a subclass:

```
TemperatureSensor ts;
ts = new Sensor(); ✘
```

It is not safe to do this because Sensor does not have any of the additional TemperatureSensor behaviour and state. Therefore we cannot invoke and use these features since the requests will fail.

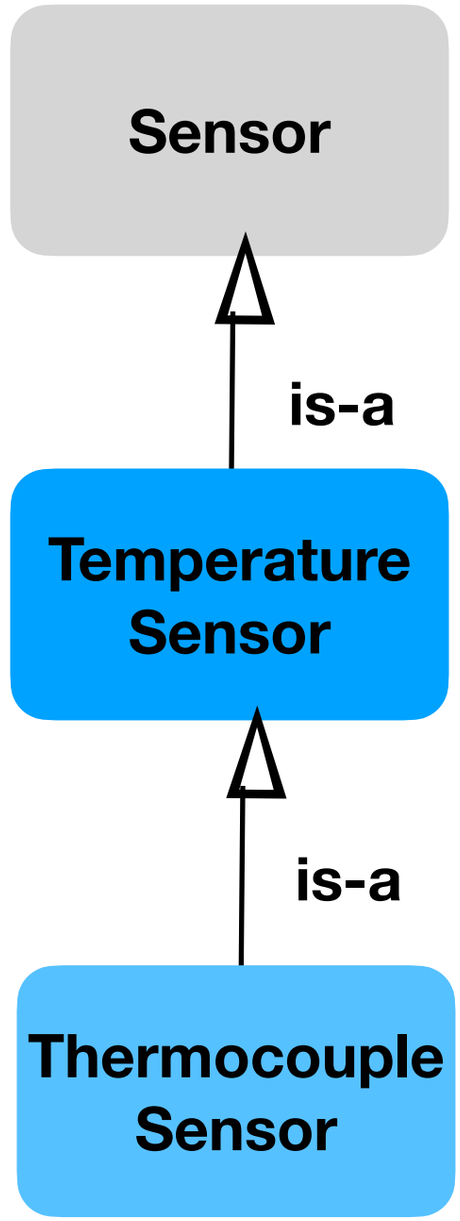
This causes a compilation error.

# Principle of Substitutability (PoS)

The principle works across an inheritance hierarchy on indirect subclasses as well.

```
Sensor s = new Sensor();
s = new ThermocoupleSensor();
//e.g. using indirect subclass

TemperatureSensor ts;
ts = new ThermocoupleSensor();
//e.g. using direct subclass
```



# Static Type vs Dynamic Type

**Static type** is the type assigned to an object variable by means of a **declaration** statement.

```
Sensor s;     // Sensor is the static type of s
```

**Dynamic type** is the actual type associated with an object variable at **run-time**.

```
s = new Sensor(); // Sensor is also the dynamic type of s
```

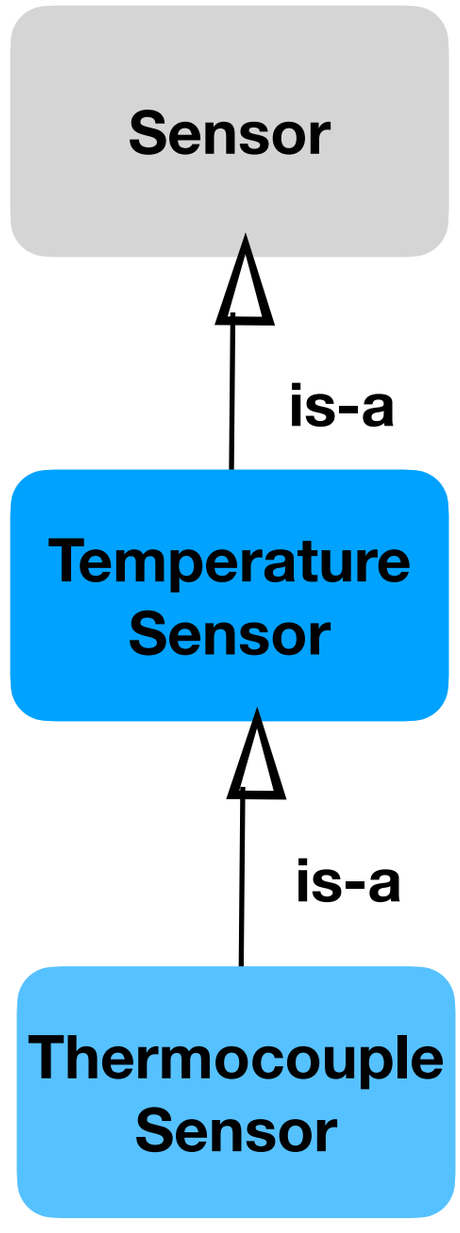
# Example - Static vs Dynamic Type

```
Sensor s = new Sensor();
//ST: Sensor,   DT: Sensor

TemperatureSensor ts = new TemperatureSensor();
//ST: TemperatureSensor,  DT: TemperatureSensor

s = new ThermocoupleSensor();
//ST: Sensor,   DT: ThermocoupleSensor
```

**(Polymorphic)**




Abbreviations:
ST - Static type;  DT - Dynamic type

# Polymorphic Objects

When the static type and the dynamic type of an object are different, that object is said to be **polymorphic.**

A polymorphic object exhibits different behaviour based on the differences between its static type and dynamic type.

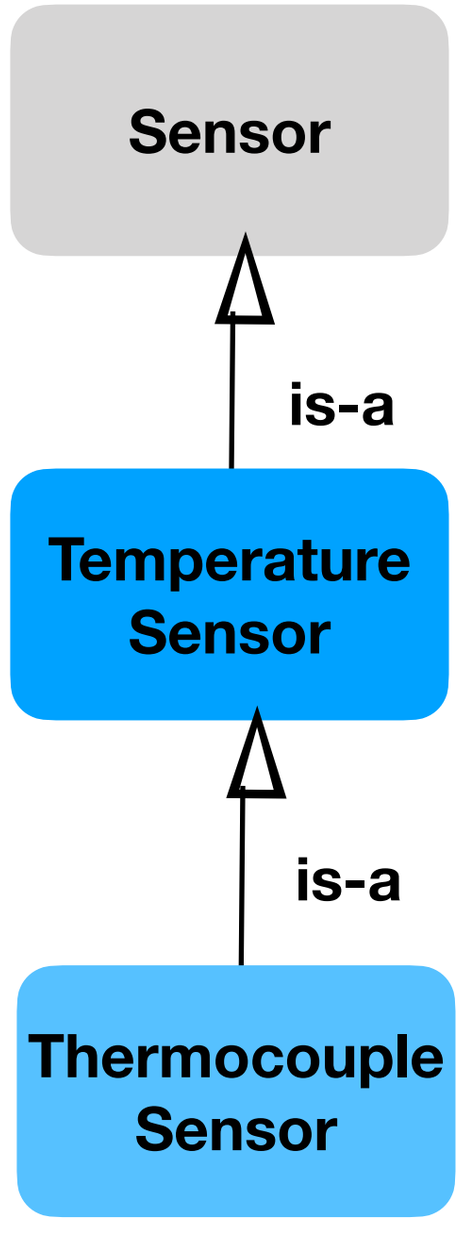
# Example - Polymorphic Objects

```
Sensor ts = new ThermocoupleSensor();
//ST: Sensor,   DT: ThermocoupleSensor

TemperatureSensor ts = new TemperatureSensor();
//ST: TemperatureSensor,   DT: TemperatureSensor

Object o = new ThermocoupleSensor();
//ST: Object,   DT: ThermocoupleSensor
```

**(Only 2 objects are Polymorphic)**




Abbreviations:
ST - Static type;  DT - Dynamic type

# Method Binding

The **static** type of an object is used to check whether a method call is permitted at **compile** time.

The **dynamic** type of an object is used to select which method is actually invoked at **run-time.**

For a polymorphic object, the static type of the object must provide the method at compile time even if the dynamic type supplies the true method that is actually used at run-time.

# Example

```
// Assume valid constructors and printout statements

Sensor s = new Sensor(15, “C”);
s.toString();


TemperatureSensor ts = new TemperatureSensor(50, “F”);
ts.toString();

ThermocoupleSensor tcs = new ThermocoupleSensor(3.41);
tcs.toString();
```

**Output:**

**Current Reading: 15C**
**50F**
**400C Thermocouple: 3.41**

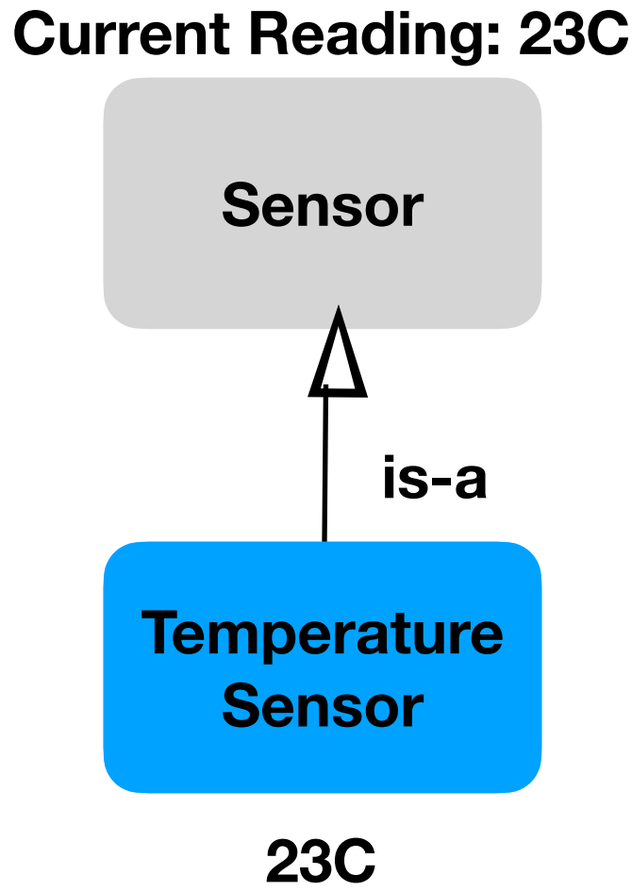
Suppose the 3 classes each have a toString( ) method that returns different values

# Example - Method Binding

```
Sensor s;
s = new TemperatureSensor(23, “C”);
s.toString();
```

**Output:**

**23C**




The Sensor class is checked for a toString( ) method at compile time. However, the TemperatureSensor’s toString( ) method is selected at run-time.

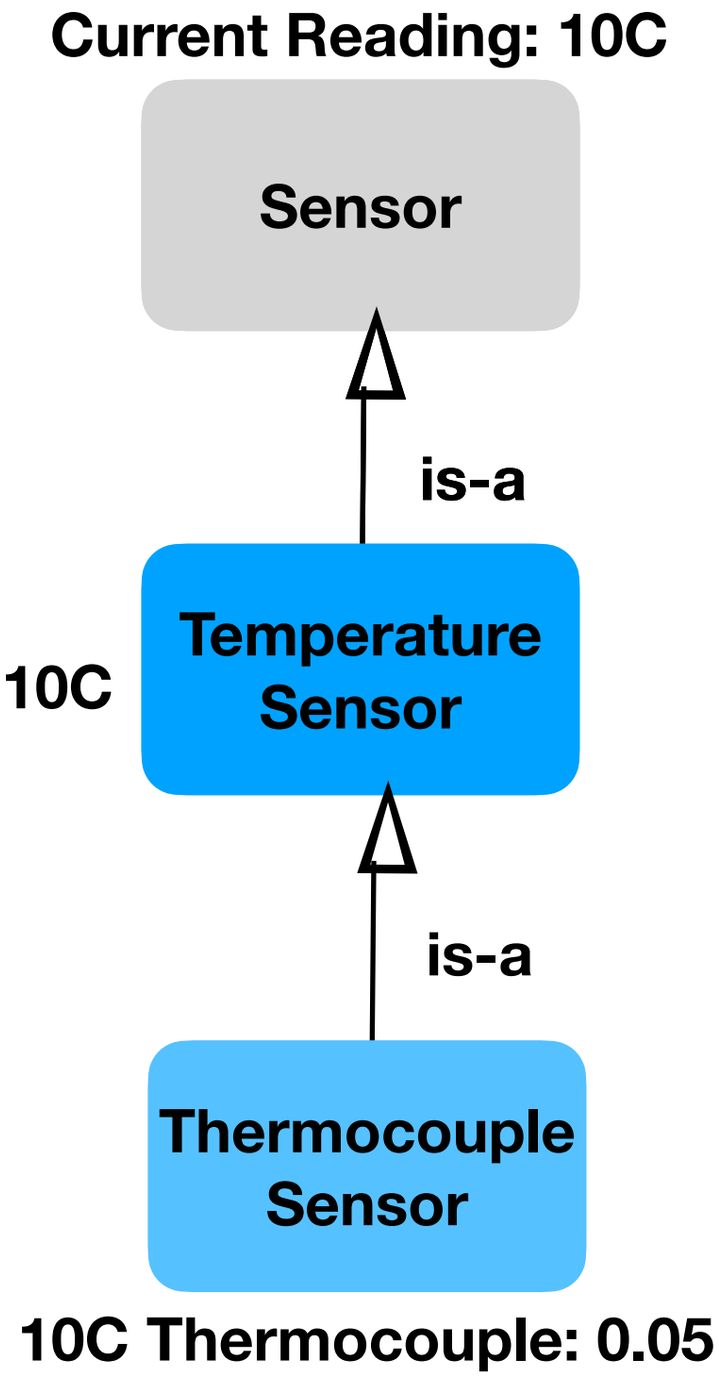
# Example - Method Binding

```
Sensor s;
s = new ThermocoupleSensor(0.05);
s.toString();
```

**Output:**

**10C Thermocouple: 0.05**

The Sensor class is checked for a toString( ) method at compile time. However, the ThermocoupleSensor's toString( ) method is selected at run-time which refines the parent TemperatureSensor toString( ) (from Week 5 slides).



# Example - Method Binding

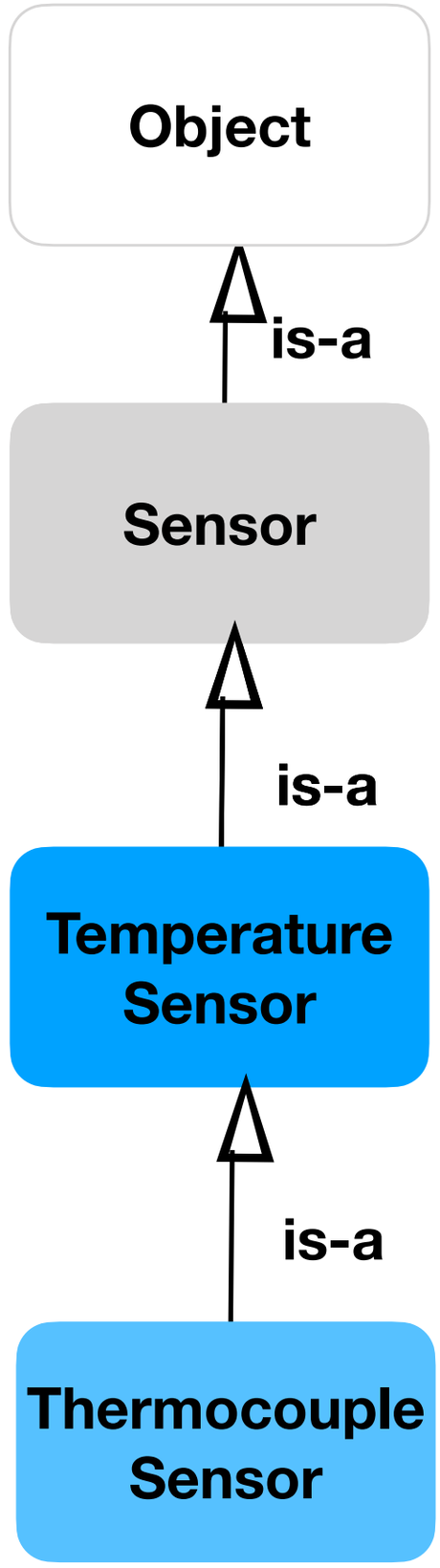
Suppose we remove all toString( ) methods from the 3 classes.

```
Sensor s;
s = new ThermocoupleSensor(0.05);
s.toString();
```

**Output:**

**ThermocoupleSensor@44ef671e**

Why does this work?

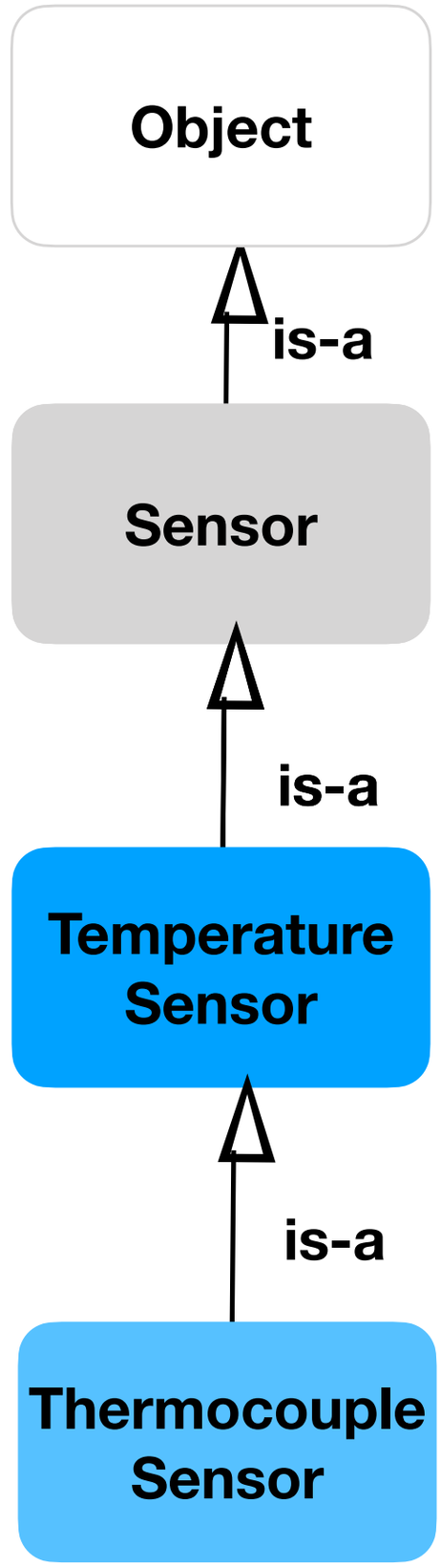


# Example - Method Binding

```
Sensor s;
s.toString();
```

**Output:**

Compilation error: NullPointerException
s has not been initialised!



# Example - Method Binding

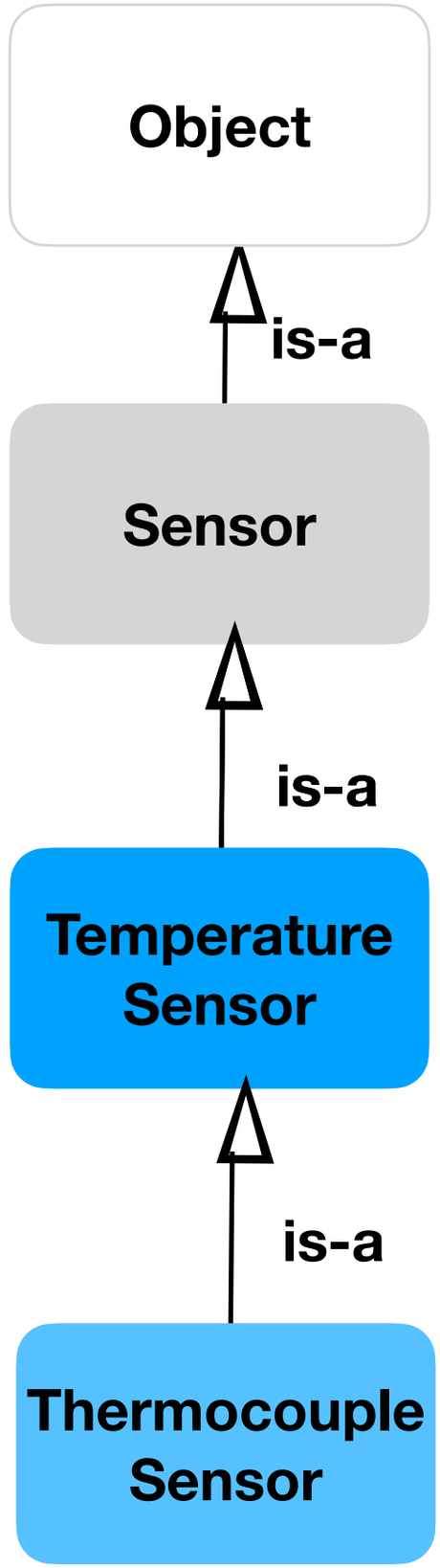
Suppose the Sensor class has a method `getCurrentValue( )` that returns the current reading of the sensor.

```
Sensor s;
s = new TemperatureSensor(23, “C”);
s.getCurrentValue();
```

**Output:**

**23C**

The search moves up the inheritance hierarchy from the dynamic type up to the static type until the method is found.



# Reverse Polymorphism Problem

The reverse polymorphism problem occurs when we have an object but we are not sure where that object lies along the inheritance hierarchy.

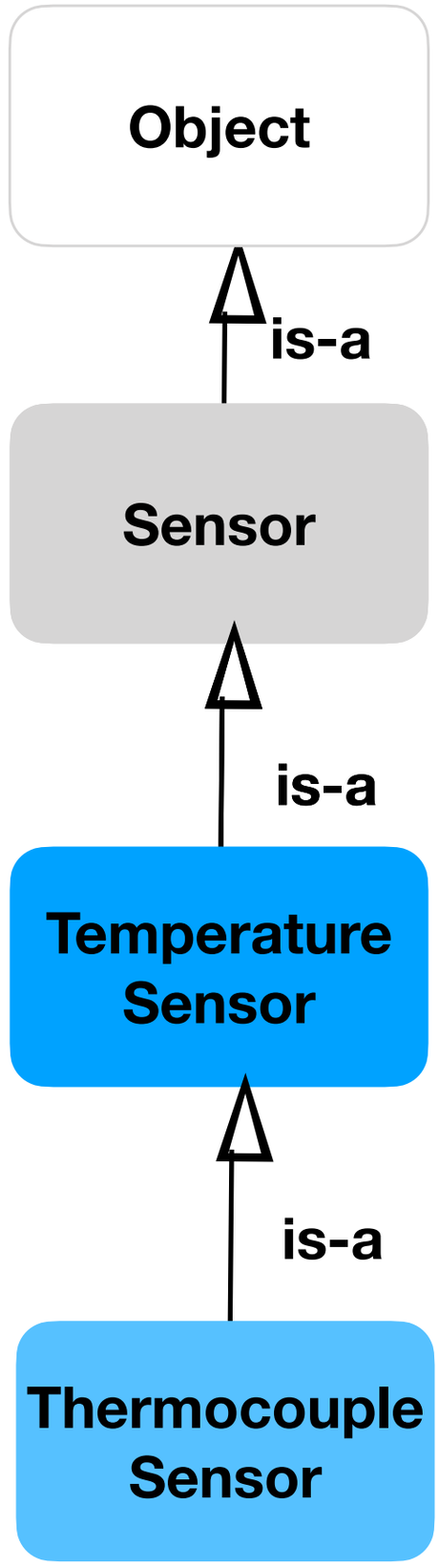
We are therefore not sure whether a method can be safely invoked.

# Example - Reverse Polymorphism Problem

```
Object s;
s = new TemperatureSensor(23, “C”);
s.getCurrentValue(); ✘
```

**Output:**

Compilation error:
No such method found in Object class!



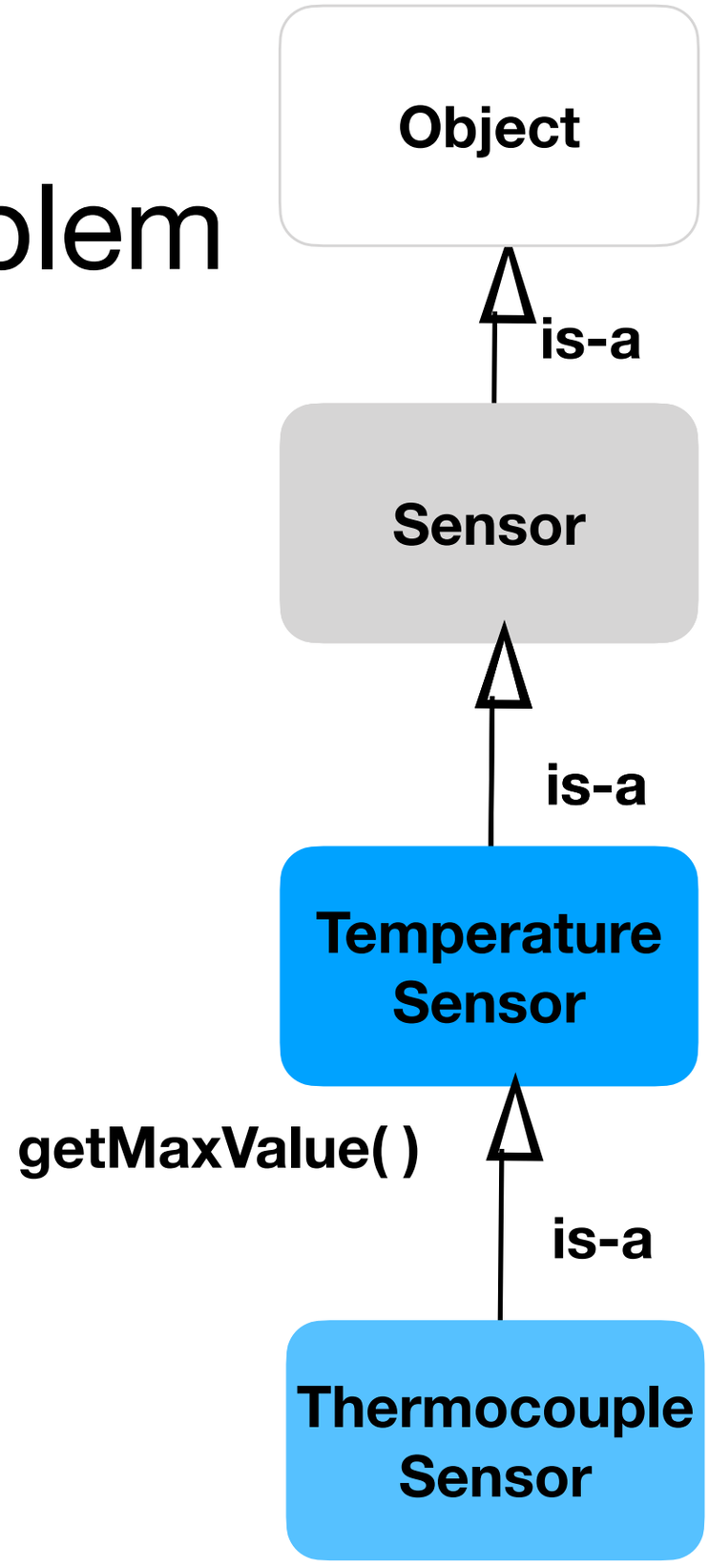
# Example - Reverse Polymorphism Problem

```
Sensor s;
s = new TemperatureSensor(23, “C”);
s.getMaxValue();
```



**Output:**

Compilation error:
No such method found in Sensor class!



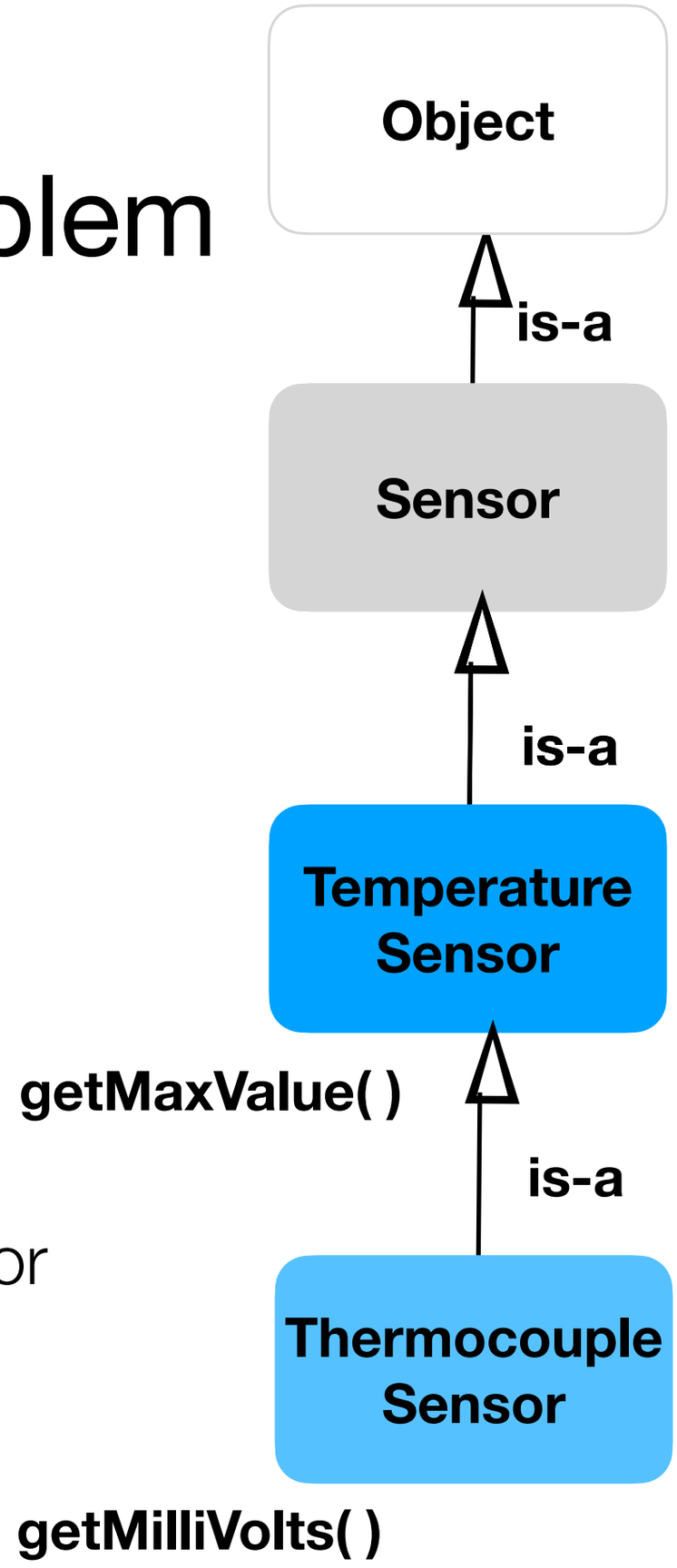
# Example - Reverse Polymorphism Problem

```
TemperatureSensor s;
s = new TemperatureSensor(23, “C”);
s.getMaxValue();
s.getMilliVolts();
```



**Output:**

Compilation error:
No such method found in TemperatureSensor class!



# Reverse Polymorphism

The reverse polymorphism problem is dealt with in two steps:

1. Use the `instanceof` operator to determine the object's true type
2. Cast the object to a **lower** type in the generalisation hierarchy

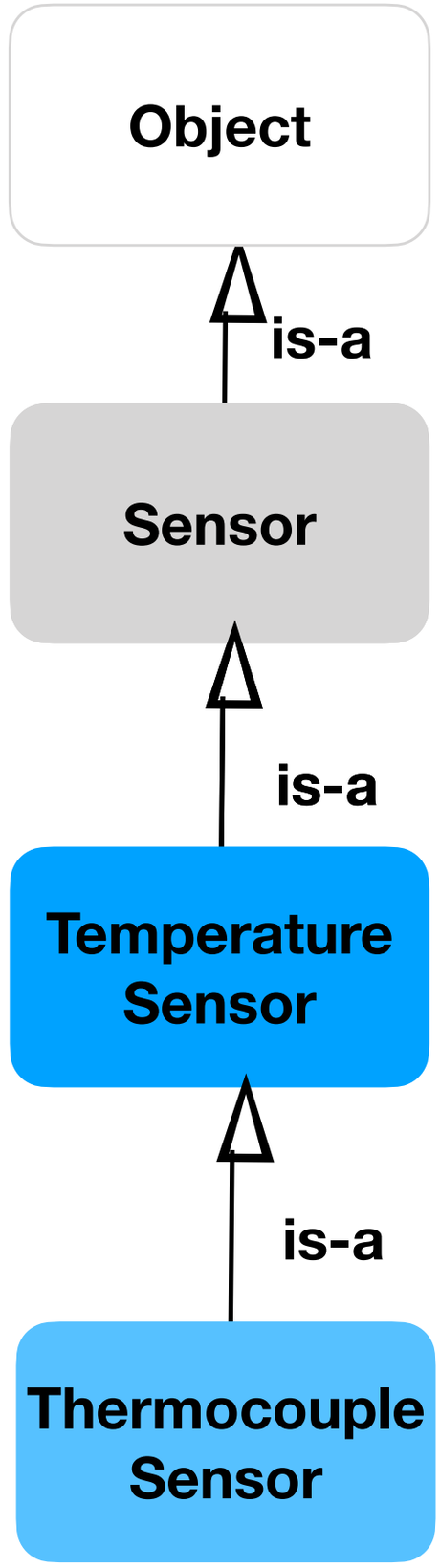
# Example - Casting Reverse Polymorphism

```
Sensor s;
s = new TemperatureSensor(23, “C”);
TemperatureSensor ts = (TemperatureSensor) s;
ts.getMaxValue();
```

**Output:**

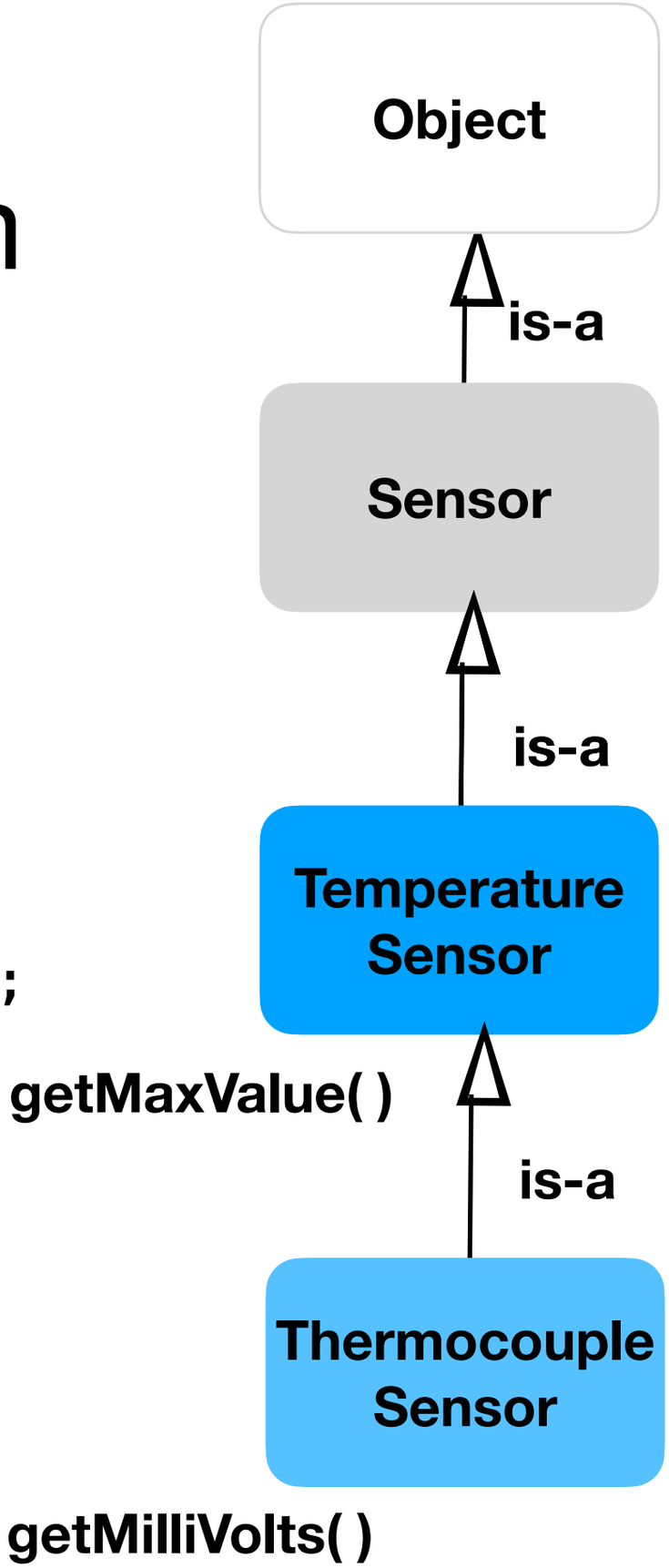
**23C**

Casting is used here since we know the type in which to cast the object to.



# Example - Casting Reverse Polymorphism

```
Sensor[] sensors = new Sensor[40];
sensors[0] = new TemperatureSensor(15,”F”);
sensors[1] = new ThermocoupleSensor(0.05);
sensors[2] = new ThermocoupleSensor(3.41);
...
sensors[39] = new TemperatureSensor(45,”C”);
```



# Example - Casting & instanceof Reverse Polymorphism

```
Sensor[] sensors = new Sensor[40];
//code adding different kinds of Sensor objects
for(Sensor s: sensors){
  if(s instanceof TemperatureSensor){
    TemperatureSensor ts = (TemperatureSensor)s;
    ts.getMaxValue();
  }
 if(s instanceof ThermocoupleSensor){
    ThermocoupleSensor tcs = (ThermocoupleSensor)s;
    tcs.getMilliVolts();
  }
}
```

# Example - Reverse Polymorphism

```
public class Device{

  // returns array of TemperatureSensor objects
  public Sensor[] getSensors(){…}

 }
```

---

```
 Device dev = new Device();

Sensor[] sensors = dev.getSensors();

 for(TemperatureSensor ts: sensors){
   ts.getMaxValue();
}
```

**Java takes care of the casting for us using the Java 5 for loop syntax**

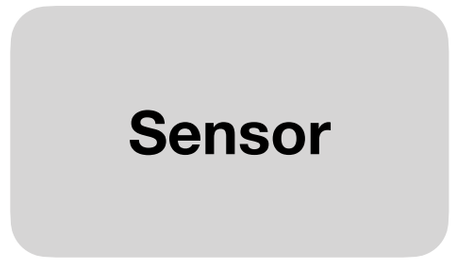
# Object Equality

The equals( ) method is meant to be overridden in all Java classes.

Object equality should be based on some conceptual evaluation of what makes two objects equal.

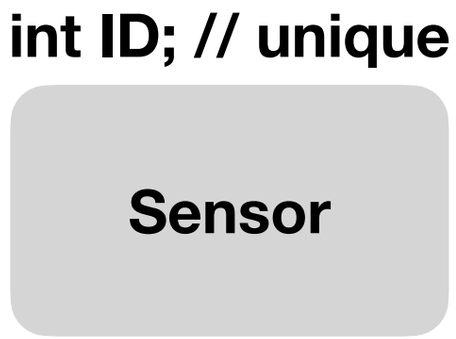
This involves some polymorphic behaviour of the objects being compared, and dealing with the reverse polymorphism problem.

# Example - equals( )




```
public boolean equals(Object obj){
    if(obj instanceof Sensor){  // obj is a Sensor
      Sensor s = (Sensor) obj;  //cast it
      if(this.ID == s.ID)     // same ID
      return true;            // objects are equal
    }
    return false;           // objects are not equal
}
```

# Example - equals( )




```
Sensor s1 = new Sensor();  // id = 10
Sensor s2 = new TemperatureSensor();  // id = 20

s1.equals(s2);   // false, IDs are different

Object o4 = new Object();
s1.equals(o4);  // false, objects are different

TemperatureSensor s3 = (TemperatureSensor)s2;
s2.equals(s3);   // true; IDs are the same (diff mem. locs)

TemperatureSensor s4 = (TemperatureSensor)s1; //compile fail
```