Objects and Classes

Encapsulation

COMP2603
Object Oriented Programming 1

Week 2, Lecture 1

Outline

- Abstraction
- Encapsulation
- Information Hiding
    - Access Modifiers
- Constructor
    - Instance Creation and Initialisation
- Message Passing Syntax
    - Method signature
    - Method overloading

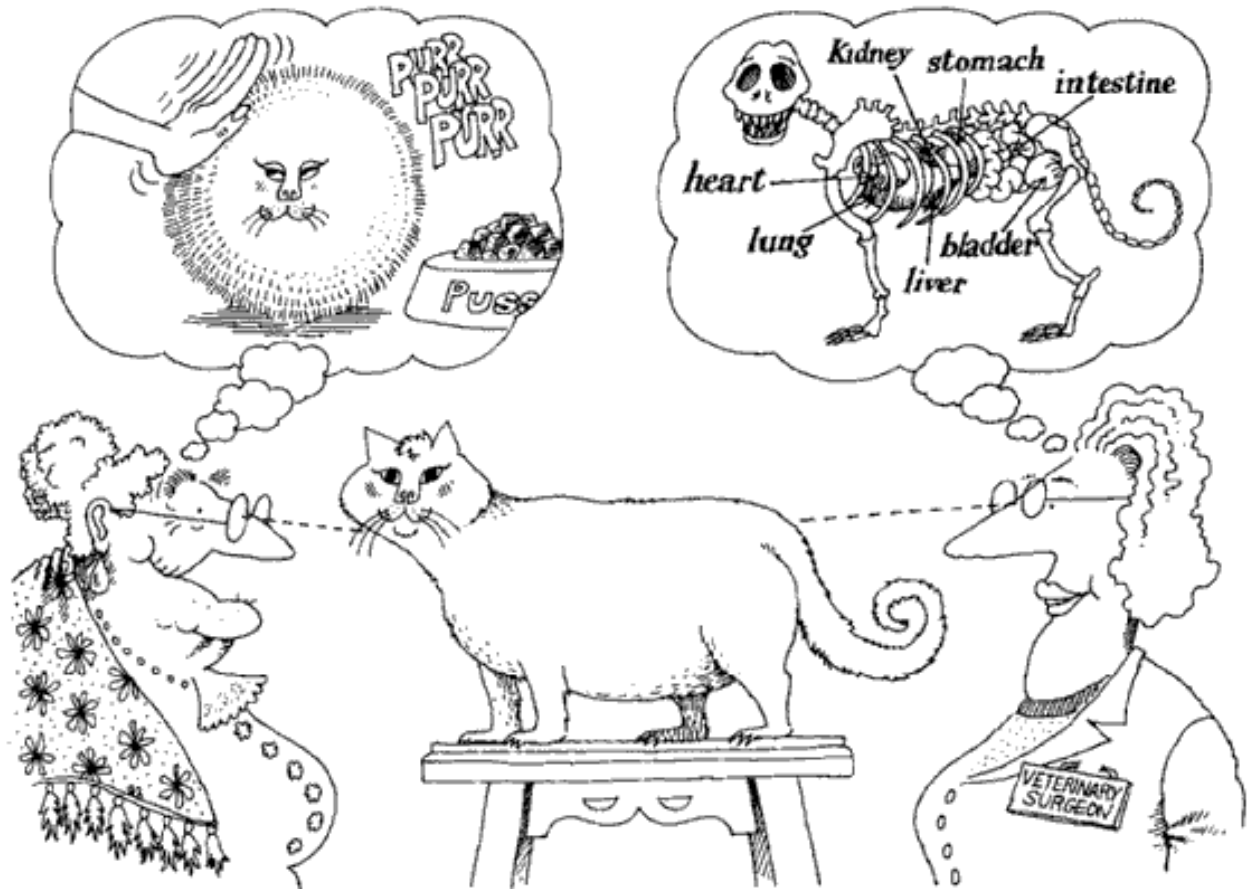
Abstraction

“Abstraction is one of the fundamental ways that we as humans cope with complexity.” (Booch, 1994).

- Recognition of similarities
- Emphasis on significant details
- Independent of implementing mechanism

Abstraction

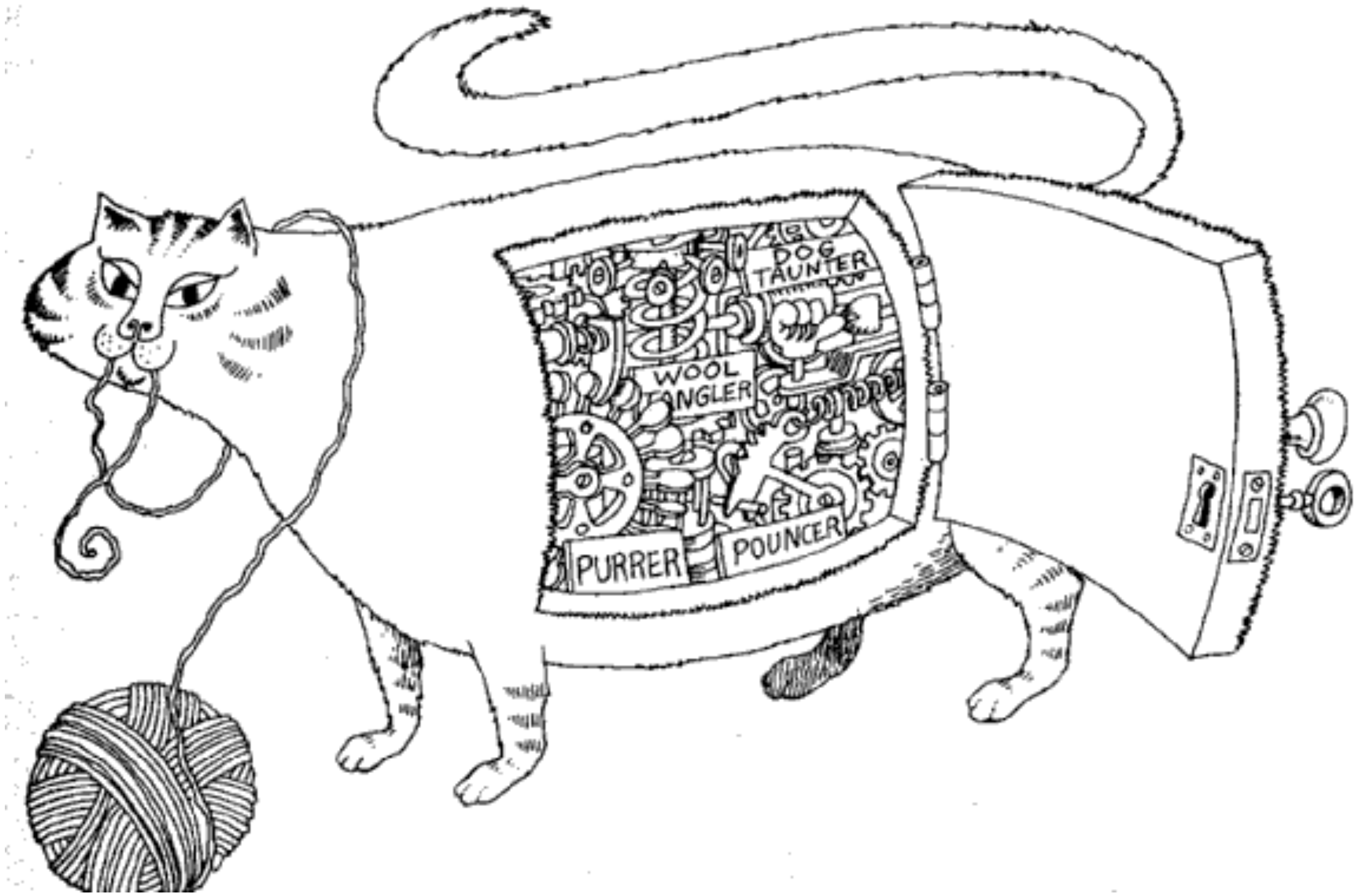
An abstraction denotes the **essential** characteristics of an object that **distinguish** it from all other kinds of objects and thus provide crisply defined **conceptual boundaries**, relative to the perspective of the viewer.



Observable behaviour

Encapsulation

Encapsulation focuses upon the implementation that gives rise to observable behaviours.



Hides the details (implementation) that gives rise to observable behaviour

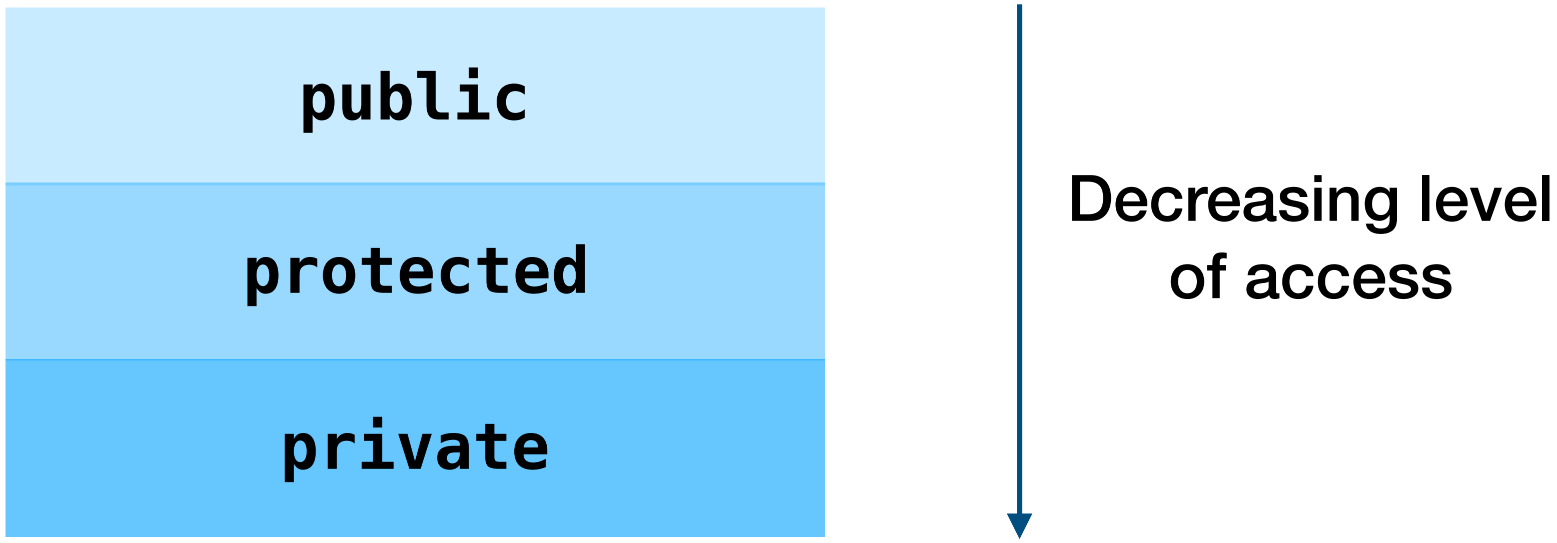
Information Hiding

Encapsulation is most often achieved through information hiding.

Information Hiding is the process of hiding all the secrets of an object that do not contribute to its essential characteristics:

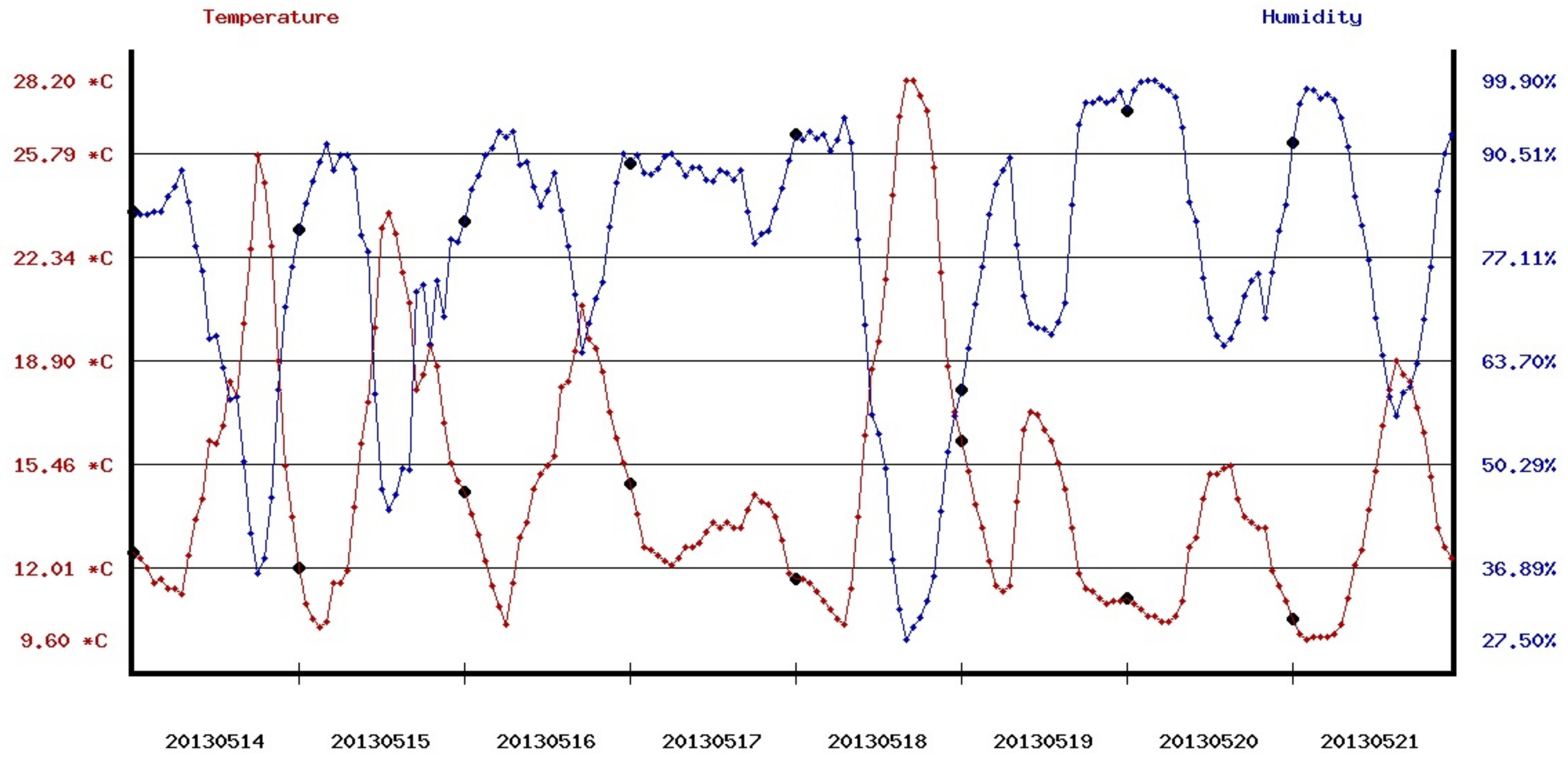
- the structure of an object is hidden
- the implementation of its methods is hidden

Access Modifiers

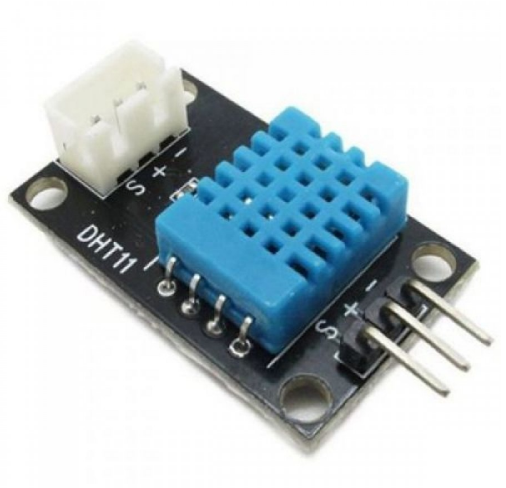


- Public: A declaration that is accessible to all clients
- Protected: A declaration that is accessible only to the class itself, its subclasses, and its friends
- Private: A declaration that is accessible only to the class itself and its friends

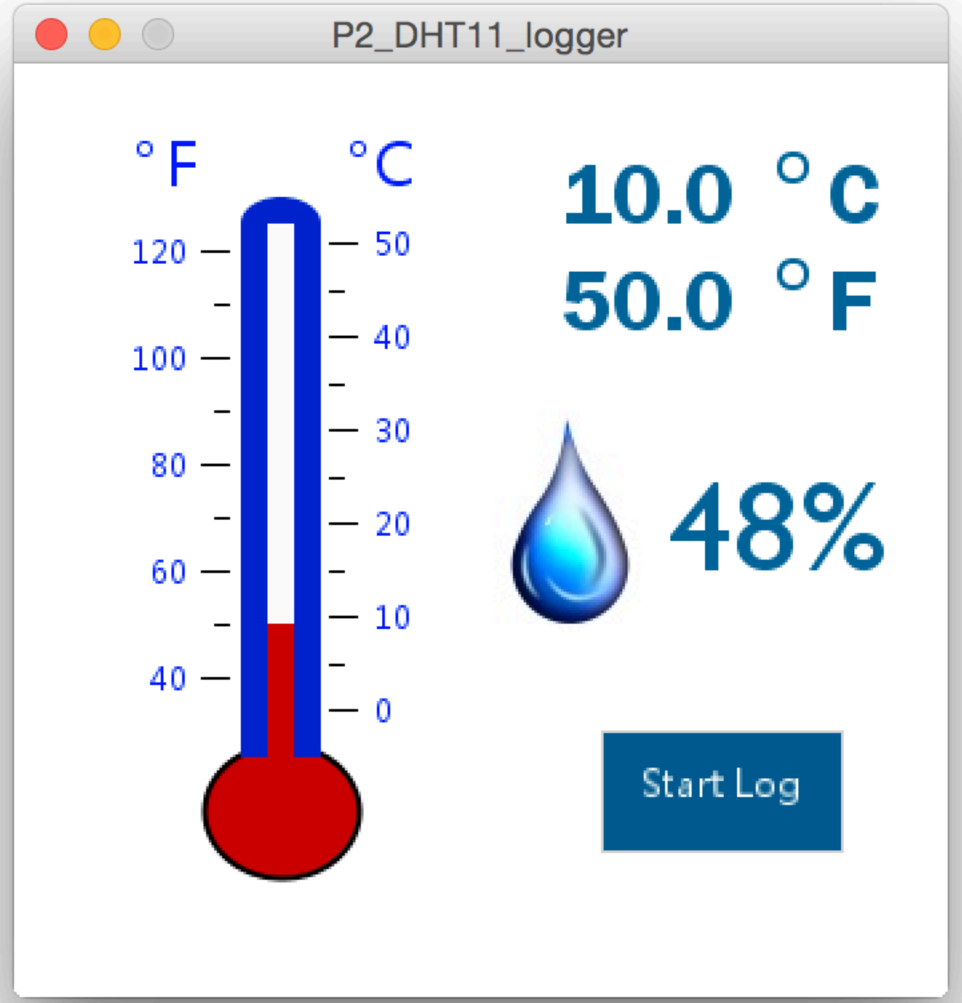
Example

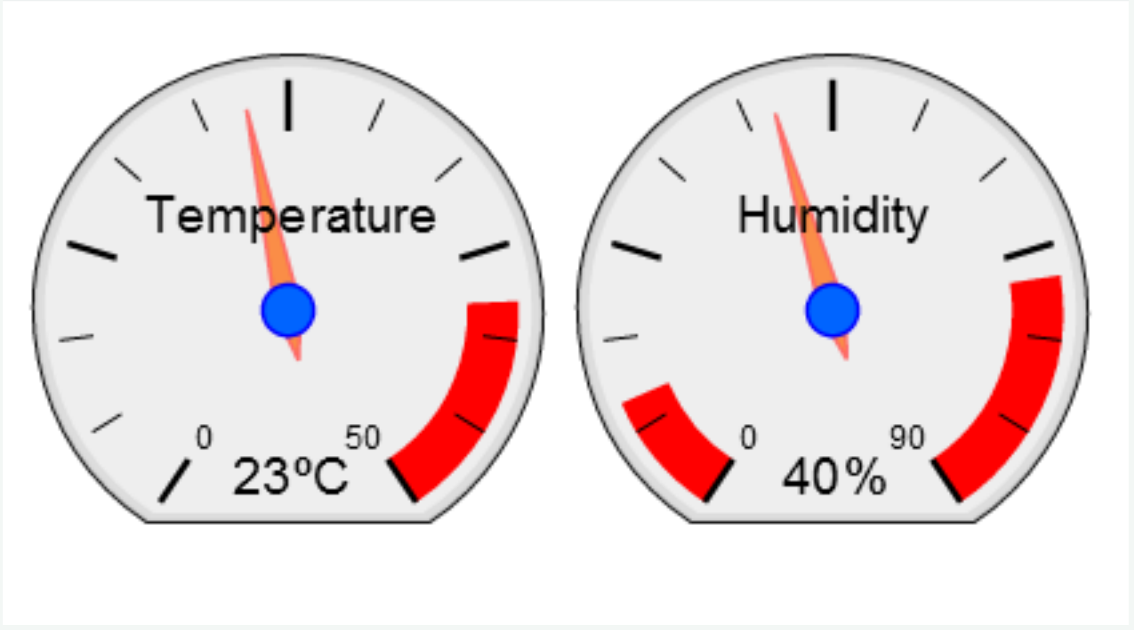


Temperature and Humidity Sensor



Example





Data visualisation

Example

```
public class TemperatureSensor{
    //State variables
    double temperature;
    final String unit = “Celcius”;//constant
}
```

t1

t2

```
t1.temperature = 100;
t2.temperature = t1.temperature;
double t2Temp = t2.temperature;
System.out.println (t2Temp + “ ”  + t2.unit);
```

Example

```
public class TemperatureSensor{
    //State variables
    private double temperature;
    private final String unit = “Celcius”;//constant
}
```

```
t1.temperature = 100;    ✗
t2.temperature = t1.temperature; ✗
double t2Temp = t2.temperature;  ✗
System.out.println (t2Temp + “ ”  + t2.unit); ✗
```

Example

```
public class TemperatureSensor{
    //State
    private double temperature;
    private final String unit = “Celcius”;

    //Accessor
    public double readTemperature( ){
      return temperature;
    }
    private void updateTemperature( ){
      /* logic for sensor to read air temperature
         and update the temperature variable */
    }
  }
```

Constructors

Constructors are used to create an object and/or initialise its state.

- Constructors always carry the same name as the class.
- Constructors do not have a return type.
- Constructors generally should have parameters for attributes that client classes may be expected to send when the instance is created.

In Java, a default constructor is provided as part of the declaration of a class. It does not take any arguments.

Constructors

In Java, a default no-argument constructor is provided as part of the declaration of a class.

It does not take any arguments.

When you create a Java class and you do not specify a constructor explicitly, the default constructor will be used when creating an instance of a class.

Example 1

Using the default no-argument constructor in Java

```
public class TemperatureSensor{
    private double temperature;
    private final String unit = “Celcius”;
    public double readTemperature( ){…}
    private void updateTemperature( ){…}
  }
```

Instances are still created

```
TemperatureSensor t1 = new TemperatureSensor();
TemperatureSensor t2 = new TemperatureSensor();
```

Example 2

Using the default no-argument constructor in Java

```
public class TemperatureSensor{
    private double temperature;
    private final String unit = “Celcius”;
    public double readTemperature( ){…}
    private void updateTemperature( ){…}
  }
```

Instances are still created

```
  TemperatureSensor[ ] sensors = new TemperatureSensor[4];
  for(int i = 0; i<4; i++)
     sensors[i] = new TemperatureSensor();
```

Constructors

If you provide a constructor of your own, the default constructor will be replaced by your customised constructor.

Example 3

```
public class TemperatureSensor{
    private double temperature;
    private final String unit;

    //Constructor explicitly typed by the programmer
    public TemperatureSensor(){

    }

  }
```

The default no-argument constructor has been replaced even though the outcome is the same. Nothing special is being done in the constructor.

Example 4

```
public class TemperatureSensor{
    private double temperature;
    private final String unit;

    //Constructor explicitly typed by the programmer
    public TemperatureSensor(){
      unit = “Celcius”;
    }

  }
```

Constructors are normally used to initialise the state of an object. Here, the unit that will used for temperature measurements is set to “Celcius”.
All TemperatureSensor objects therefore will have the default value of “Celcius” when instantiated.

Example 5

```
public class TemperatureSensor{
    private double temperature;
    private final String unit;

    //Constructor that requires an argument
    public TemperatureSensor(String desiredUnit){
      unit = desiredUnit;
    }

  }
```

TemperatureSensor objects can now have any unit specified when instantiated.

Example 5 continued

```
TemperatureSensor t1 = new TemperatureSensor(“Celcius”);
TemperatureSensor t2 = new TemperatureSensor(“Farenheit”);
TemperatureSensor t3 = new TemperatureSensor(“Kelvin”);
```

Three different TemperatureSensor objects each with different units

```
TemperatureSensor t4 = new TemperatureSensor(“Kevin”);
```

Typo! How can this be avoided?

Example 6

```
public class TemperatureSensor{
    private double temperature;
    private final String unit;

    //Constructor that requires an argument
    public TemperatureSensor(String desiredUnit){
       if( (value.equals(“Fahrenheit”))    ||
           (value.equals(“Kelvin”))){
             unit = value;
         }
      else
         unit = “Celcius”;
    }
  }
```

Error checking of argument

Example 7

```
public class TemperatureSensor{
    private double temperature;
    private final String unit;

    //Constructor that requires an argument
    public TemperatureSensor(String desiredUnit){
       setUnit(desiredUnit);   // use the mutator
     }
    private void setUnit(String value){
       if( (value.equals(“Celcius”)) ||
       (value.equals(“Fahrenheit”))
          unit = value;
       else
          unit = “Celcius”;
    }
  }
```

A better way of doing this since constructors should simply initialise rather than have extended logic

Example 7 continued

```
TemperatureSensor t4 = new TemperatureSensor(“Kevin”);
```

The units would be set to “Celcius” now.

How can we check?

Example 8

```
public class TemperatureSensor{
     private double temperature;
     private final String unit;

     //Constructor that requires an argument
     public TemperatureSensor(String desiredUnit){
        setUnit(desiredUnit);   // use the mutator
      }
     private void setUnit(String value){
        if( (value.equals(“Celcius”)) || (value.equals(“Fahrenheit”))
            unit = value;
        else
            unit = “Celcius”;
     }
     public double getUnit( ){
        return unit;
     }
   }
```

Accessor for the unit variable

Example 8 continued

```
TemperatureSensor t4 = new TemperatureSensor(“Kevin”);
String unit = t4.getUnit();
System.out.println(unit);
```

Output:

```
Celcius
```

Methods

A method causes certain actions to take place

Method Overloading

The process of writing methods in the same class with the same name but with different method signatures.

These methods are said to be overloaded.

Example

```
public class TemperatureSensor{
    //State
    private double temperature;
    private final String unit = “Celcius”;

    //Accessor
    public double readTemperature( ){
       return temperature;
    }
    public double readTemperature(String units){
       if(units.equals(“Fahrenheit”)
          return temperature * 9/5 + 32 ;
       else
          return temperature
    }
  }
```

References

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