Abstract Classes & Interfaces

COMP2603
Object Oriented Programming 1

Week 5

Outline

- Abstract classes and methods
- Preventing Inheritance: Final classes and methods
- Interfaces

Abstract Classes

An abstract class is one that cannot be instantiated.

It is useful when we want to factor out common attributes and behaviour from several classes.

Subclasses of an abstract class can then inherit these common attributes and behaviour.

This simplifies the design, development and maintenance of the subclasses.

Creating an Abstract Class

A class is made abstract using the keyword abstract in the class declaration as follows:

```
public abstract class Sensor{
    // class body as before
}
```

Instantiating an Abstract Class

When the Sensor class is made abstract, we can no longer create instances of the Sensor class as follows:

```
Sensor s = new Sensor( );
```

This causes a compilation error because abstract classes cannot be instantiated using the `new` keyword.

Abstract vs Concrete

Terminology:

- **Abstract class:** cannot create an instance of the class
- **Concrete class:** instances can be created (new)

- **Abstract method:** only has a method declaration. No method body
- **Concrete method:** has a method declaration and a method body

Abstract Methods

An abstract method in a superclass enforces a protocol in all of its subclasses without specifying how the protocol is to be implemented.

All subclasses of an abstract method therefore must provide an implementation of that method.

The abstract class does not care how the abstract methods are implemented in a subclass. The only thing that matters is that the subclass provides a method implementation according to the contract laid out by the abstract class in the abstract method signature.

Creating an Abstract Method

An abstract method takes the following form:

```
access-modifier abstract return-type method-
Name( parameter-type parameter-name);
```

Examples: In the Sensor class

```
public abstract void measure( );
public abstract boolean testWorking( );
```

Subclasses of Abstract Classes

Subclasses would therefore need to provide a method body for all abstract methods as follows

```
public class HumiditySensor extends Sensor{
    public void measure( ){
      //code to measure humidity
    }
    public boolean testWorking(){
      // code to return T/F if the HumiditySensor works
    }
}
```

Subclasses of Abstract Classes

These methods are now **concrete methods** in the subclasses.

```
public class LightSensor extends Sensor{
  public void measure( ){
    //code to measure light intensity
  }
  public boolean testWorking(){
    // code to return T/F if the LightSensor works
  }
}
```

Subclasses of Abstract Classes

These methods can now be invoked on instances of types that along the inheritance hierarchy starting from the parent (Sensor) until the child (DistanceSensor, LightSensor, etc)

```
public class DistanceSensor extends Sensor{
    public void measure( ){
      //code to measure distances
    }
    public boolean testWorking(){
      // code to return T/F if the DistanceSensor works
    }
}
```

Example

```
1. LightSensor ls = new LightSensor();
2. HumiditySensor hs = new HumiditySensor();
3, DistanceSensor ds = new DistanceSensor();


   ls.testWorking();
   hs.testWorking();
   ds.testWorking();



   ls.measure();
   hs.measure();
   ds.measure()
```

Example

```
Sensor ls = new LightSensor();
Sensor hs = new HumiditySensor();
Sensor ds = new DistanceSensor();

ls.testWorking();
hs.testWorking();
ds.testWorking();

ls.measure();
hs.measure();
ds.measure()
```

We can refer to the objects using the parent type thereby creating polymorphic variables

Example

```
Sensor[] sensors = new Sensor[5];
sensors[0] = new ThermocoupleSensor();
sensors[1] = new LightSensor();
sensors[2] = new HumiditySensor();
sensors[3] = new MoistureSensor();
sensors[4] = new DistanceSensor();
```

```
for(int i = 0; i < sensors.length; i++){
   Sensor s = sensor[i];
   s.testWorking(); //each subclass defines how to test if it works
   s.measure(); // each subclass takes measurements in its own way
}
```

Example

```
Sensor[] sensors = new Sensor[5];

sensors[0] = new ThermocoupleSensor();
sensors[1] = new LightSensor();
sensors[2] = new HumiditySensor();
sensors[3] = new MoistureSensor();
sensors[4] = new DistanceSensor();
```

```
for (Sensor s: sensors){ // iterates the same as the previous for loop
  s.testWorking();  // each subclass defines how to test if it works
  s.measure(); // each subclass takes measurements in its own way
 }
```

Preventing Overriding

Child classes can be prevented from overriding a parent's inherited method.

This means that the method implementation in the parent class will always be used whenever it is called on any child instance.

Careful consideration must be done since all child classes would have that particular behaviour.

Design Pattern - Template

Preventing Overriding

A final method takes the following form:

```
access-modifier final return-type method-Name( parameter-type parameter-name);
```

Example: In the Sensor class

```
public final void turnOn( ){
  // code to turn on sensor
}
```

Preventing Inheritance

Inheritance of an entire class can be prevented by using the final keyword as well.

This means that the class will not be subclassed at all.

Preventing Inheritance

A final class takes the following form:

```
access-modifier final class className
```

Example:

```
public final class UltrasonicSensor extends Sensor{
 // UltrasonicSensor class body
}
```

Interface

An interface makes it possible to deal with objects which are not known except that they implement a minimum set of behaviours specified by the interface.

Example (open and close):

A door

A book

A bank account

Common behaviours that take place in different ways.

Defining an Interface

```
public interface Openable{
   public abstract void open();
   public abstract void close();
}
```

Implementing an Interface

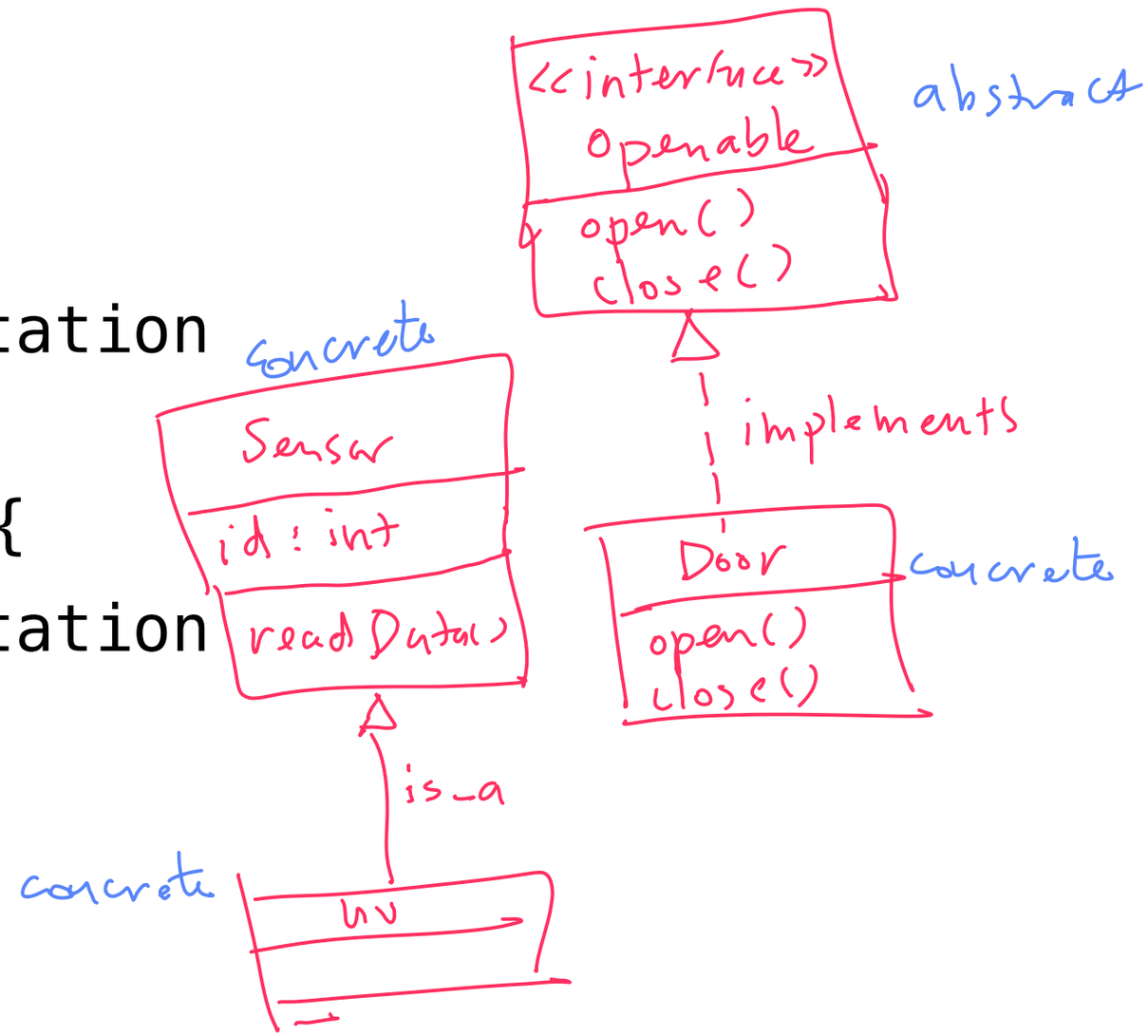
Subtypes

```
public class Door implements Openable{

  // door methods

  public void open(){
    //method implementation
  }
  public void close(){
    //method implementation
  }

}
```



Implementing an Interface

```
public class Account implements Openable{

  // account methods

  public void open(){
    //method implementation
  }
  public void close(){
    //method implementation
  }

}
```

Implementing an Interface

```
public class Book implements Openable{

  // book methods

  public void open(){
    //method implementation
  }
  public void close(){
    //method implementation
  }

}
```

Example

```
Openable[ ] objects = new Openable[3];
objects[0] = new Door();
objects[1] = new Book();
objects[2] = new Account();
```

```
for(Openable obj: objects){
    obj.open();
    obj.close();
}
```

Exercises

Would these statements work? Why or why not?

START : 11:32
END : 11:42

```
Y Sensor uls = new UltrasonicSensor();  //1
N Sensor s = new Sensor();              //2
N TemperatureSensor ts = new Sensor( ); //3
Y DistanceSensor ds = new DistanceSensor(); //4
N LightSensor ls = new HumiditySensor();    //5
W HumiditySensor[] hs = new Sensor[4];   //6
Y Sensor[] s = new Sensor[8];            //7
```