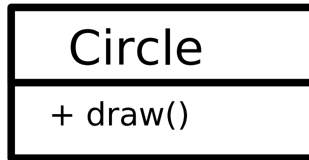
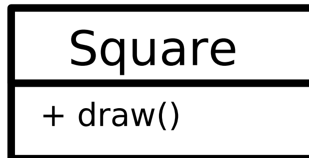
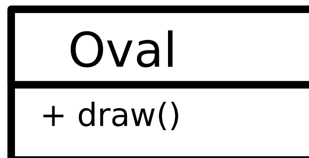
Modifiers

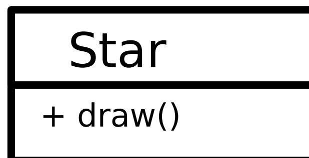
	Class	Package	Subclass	World
public	✓	✓	✓	✓
protected	✓	✓	✓	✗
package	✓	✓	✗	✗
private	✓	✗	✗	✗

The Canonical Example I



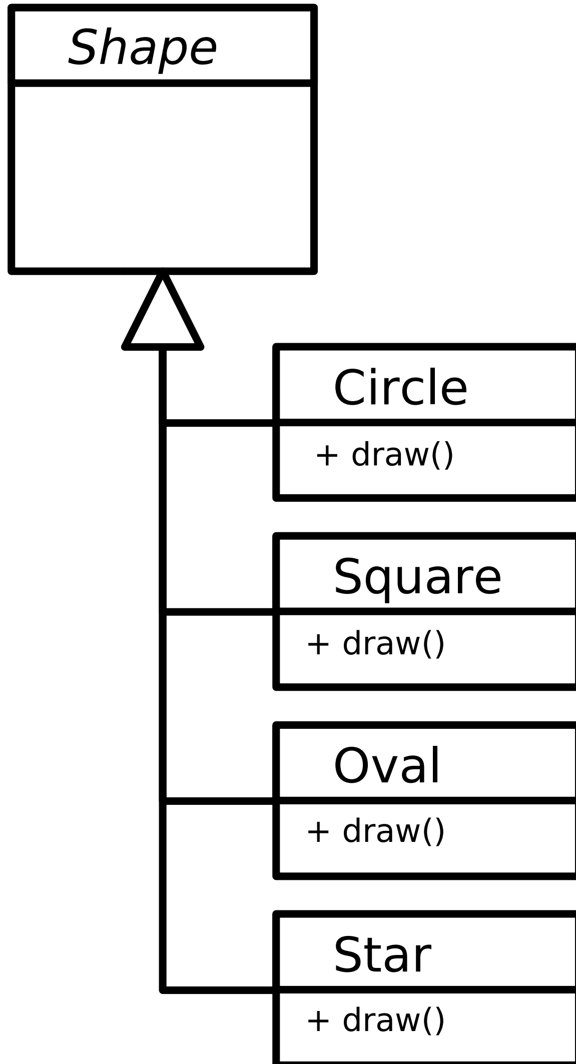






- A drawing program that can draw circles, squares, ovals and stars
- It would presumably keep a list of all the drawing objects
- **Option 1**
  - Keep a list of Circle objects, a list of Square objects,...
  - Iterate over each list drawing each object in turn
  - What has to change if we want to add a new shape?

The Canonical Example II

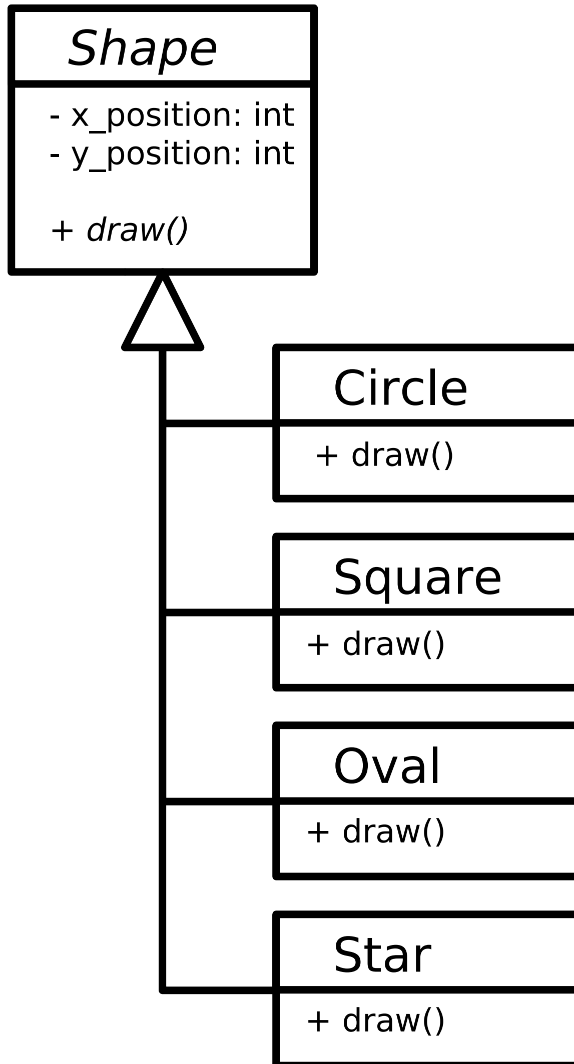


- **Option 2**
  - Keep a single list of Shape references
  - Figure out what each object really is, narrow the reference and then draw()

```
for every Shape s in myShapeList
   if (s is really a Circle)
      Circle c = (Circle)s;
      c.draw();
   else if (s is really a Square)
      Square sq = (Square)s;
      sq.draw();
   else if...
```

  - What if we want to add a new shape?

The Canonical Example III



- **Option 3 (Polymorphic)**
  - Keep a single list of Shape references
  - Let the compiler figure out what to do with each Shape reference

```
For every Shape s in myShapeList
  s.draw();
```

  - What if we want to add a new shape?

Implementations

- Java
  - All methods are dynamic polymorphic.
- Python
  - All methods are dynamic polymorphic.
- C++
  - Only functions marked *virtual* are dynamic polymorphic

- Polymorphism in OOP is an extremely important concept that you need to make <u>sure</u> you understand...

Abstract Methods

NOT JAVA

```
class Person {
  public void dance();
}

class Student extends Person {
  public void dance() {
    body_pop();
  }
}

class Lecturer extends Person {
  public void dance() {
    jiggle_a_bit();
  }
}
```

- There are times when we have a definite concept but we expect every specialism of it to have a different implementation (like the draw() method in the Shape example). We want to enforce that idea without providing a default method
- E.g. We want to enforce that all objects with Person in their ancestry support a dance() method
  - But there isn't now a default dance()
- We specify an **abstract** dance method in the Person class
  - i.e. we don't fill in any implementation (code) at all in Person.

Abstract Classes

- Before we could write Person p = new Person()
- But now p.dance() is undefined
- Therefore we have implicitly made the class abstract ie. It cannot be directly instantiated to an object
- Languages require some way to tell them that the class is meant to be abstract and it wasn't a mistake:

```
public abstract class Person {
   public abstract void dance();
}
```

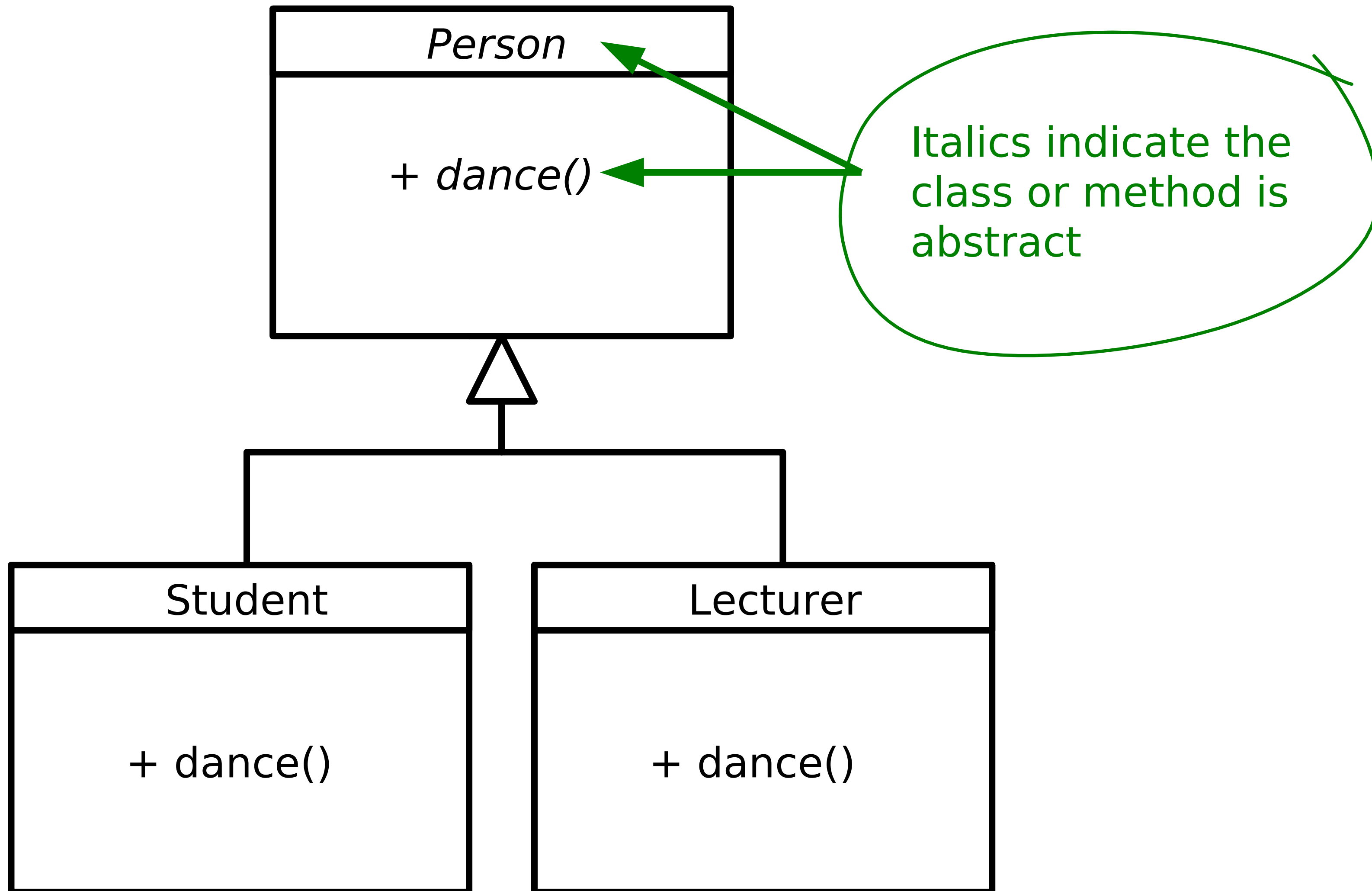
Java

```
class Person {
   public:
      virtual void dance()=0;
}
```

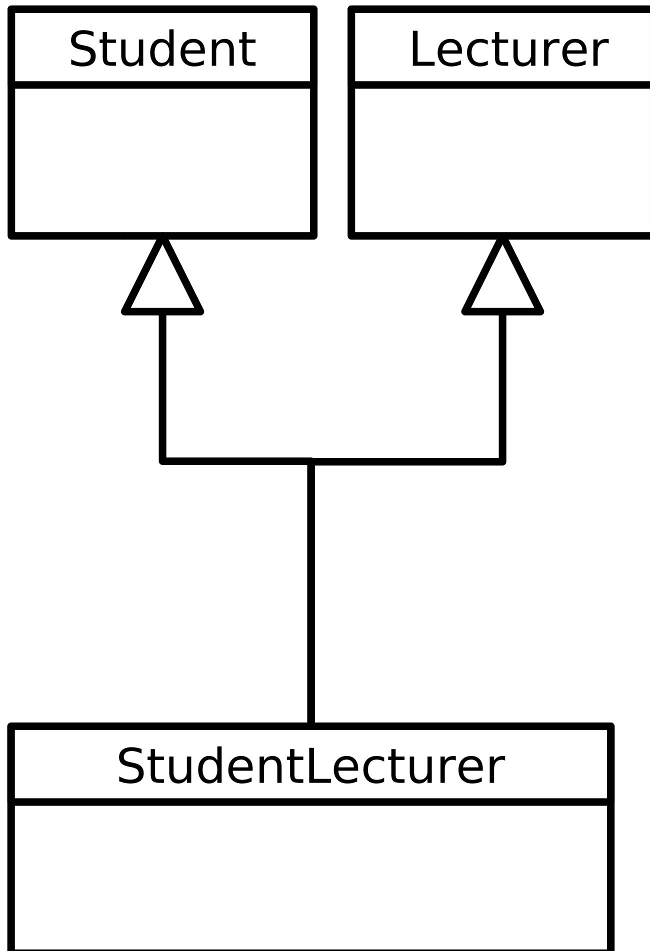
C++

- Note that an abstract class can contain state variables that get inherited as normal
- Note also that, in Java, we can declare a class as abstract despite not specifying an abstract method in it!!

Representing Abstract Classes

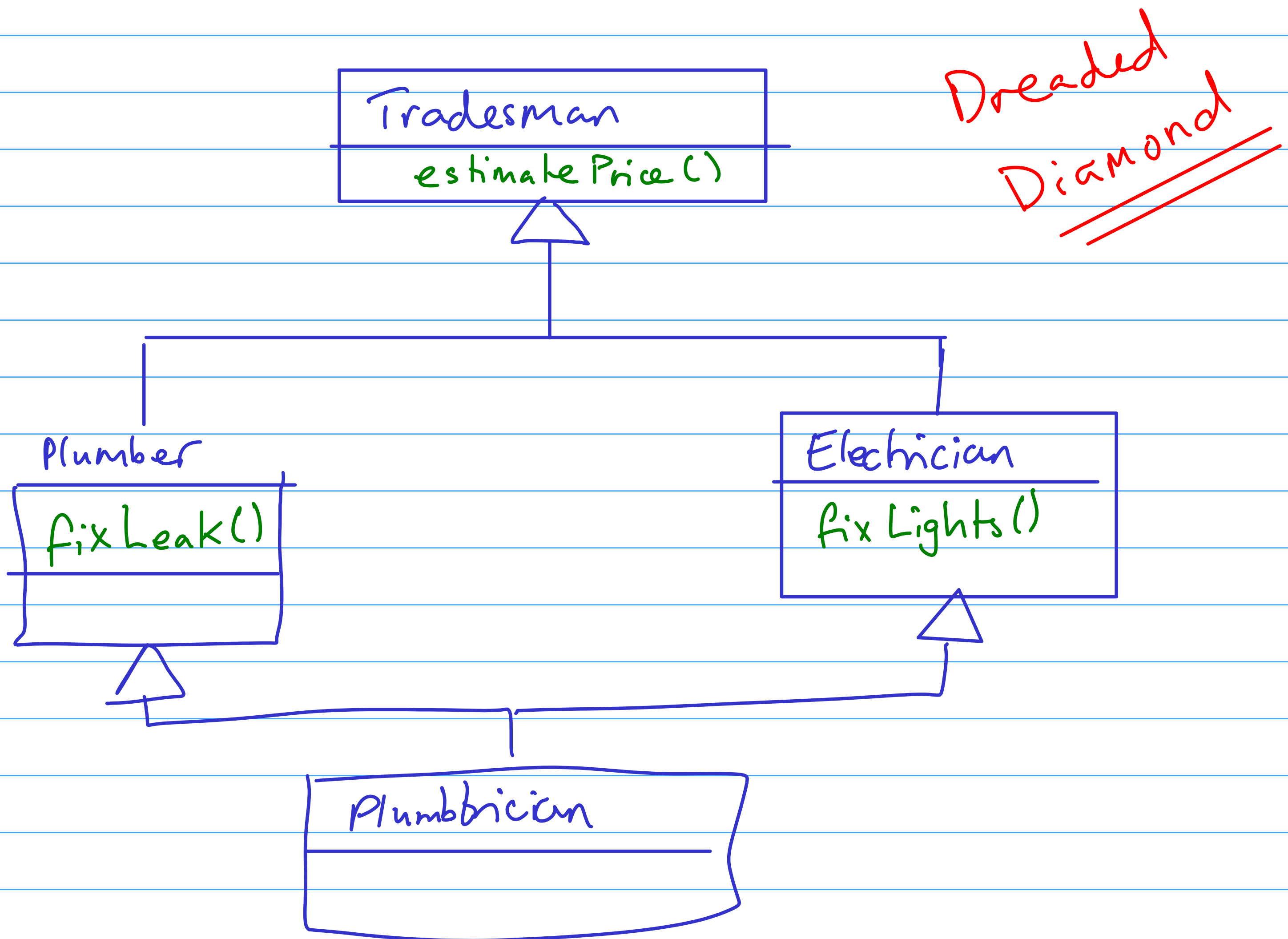


Multiple Inheritance



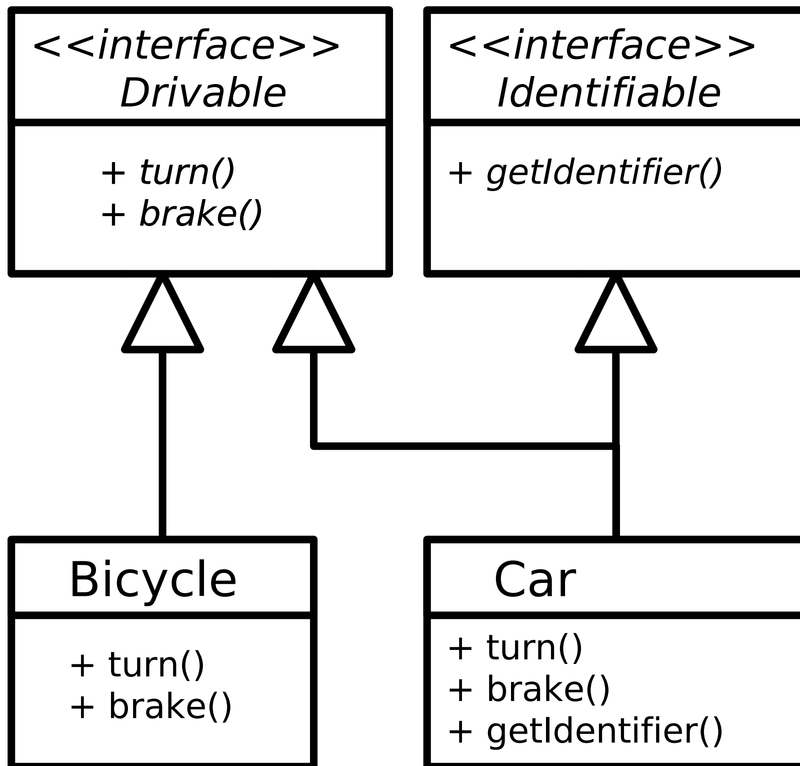
- What if we have a Lecturer who studies for another degree?
- StudentLecturer inherits two dance() methods... which does it do?
- The Java solution? **You can only extend (inherit) from <u>one</u> class in Java**
  - (which may itself inherit from another...)
  - This is Java-specific (C++ allows multiple class inheritance)

Plumber / Electrician Example



Interfaces (Java only)

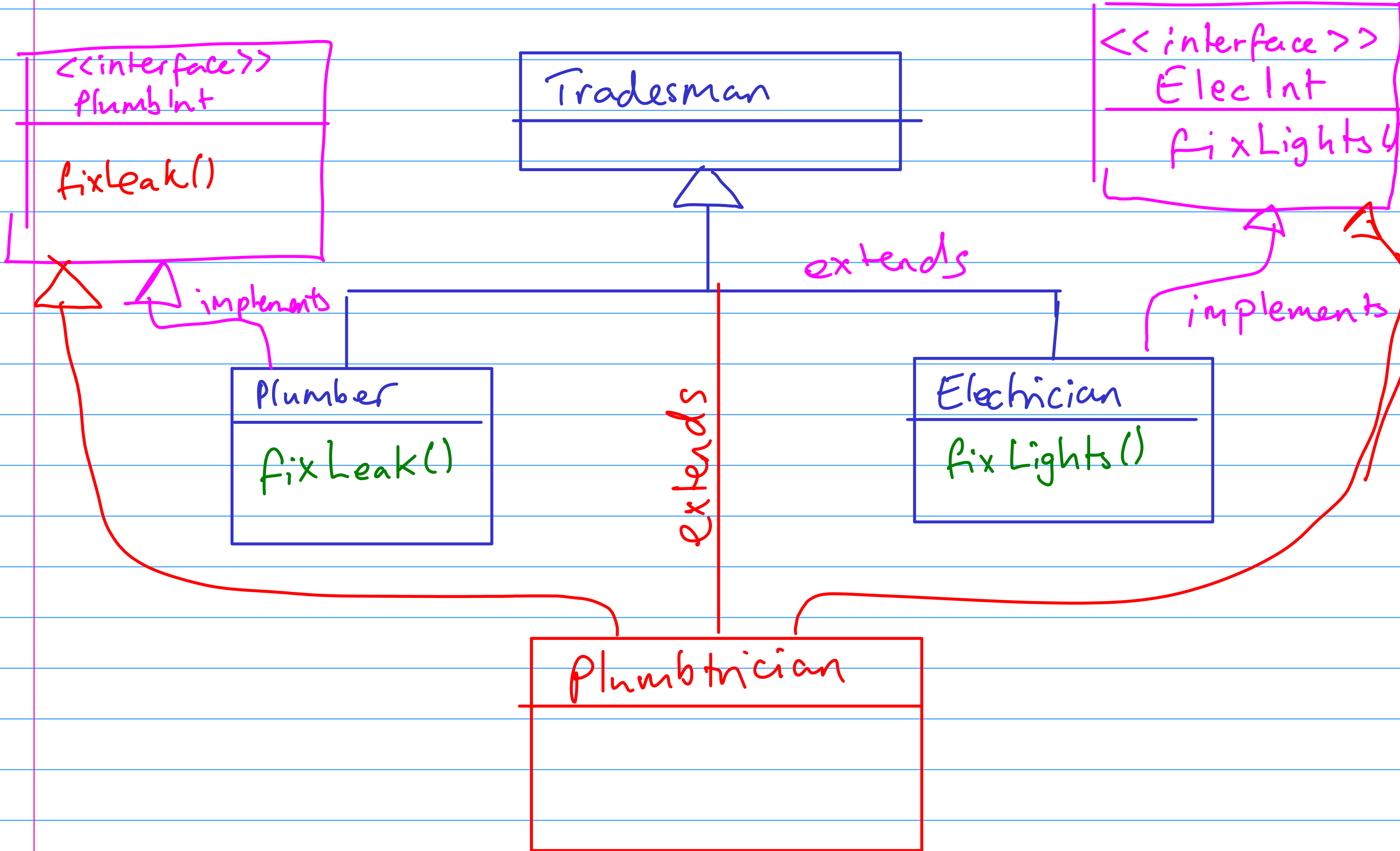
- Java has the notion of an **interface** which is like a class but with **no state** and **all methods abstract**
- For an interface, there can then be no clashes of methods or variables to worry about, so we can allow multiple inheritance



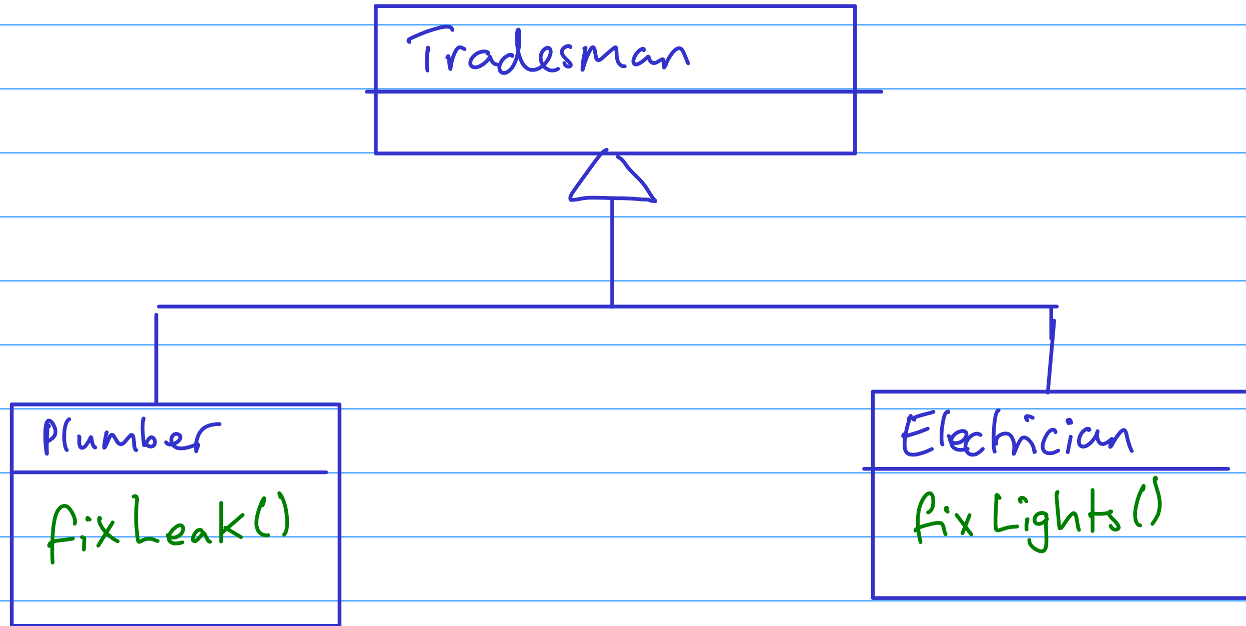

```
Interface Drivable {
  public void turn();
  public void brake();
}

Interface Identifiable {
  public void getIdentifier();
}

class Bicycle implements Drivable {
  public void turn() {...}
  public void brake() {... }
}

class Car implements Drivable, Identifiable {
  public void turn() {...}
  public void brake() {... }
  public void getIdentifier() {...}
}
```

abstract assumed for interfaces

Fixing With Interfaces



Fixing With Interfaces



Recap

- Important OOP concepts you need to understand:
  - Modularity (classes, objects)
  - Data Encapsulation
  - Inheritance
  - Abstraction
  - Polymorphism