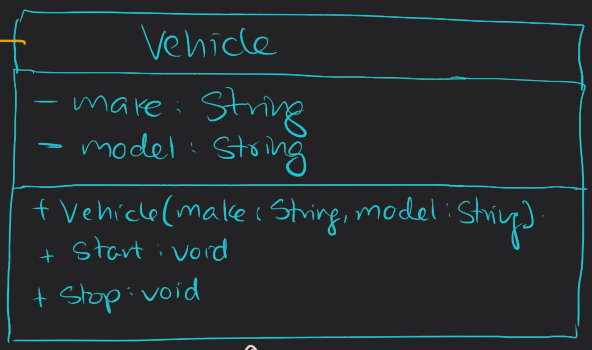
INHERITANCE



Base/Parent class

"is a" relationship

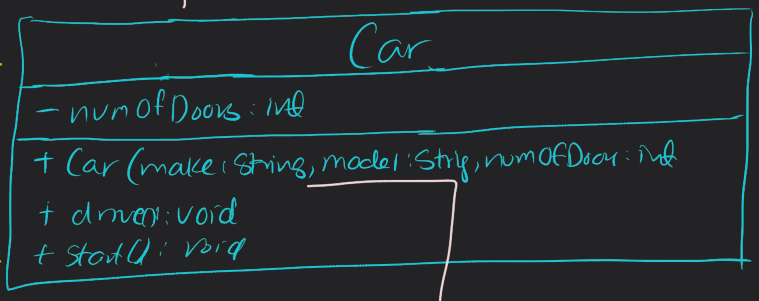
So Car has access to Vehicle public stuff directly

extends/inherits

Subclass/child class

- can have its own attributes & methods

will override




```
public Car(String make, String model, int numOfDoors) {
    super(make, model);
    this.numOfDoors = numOfDoors;
}
```

calls the constructor of superclass "Vehicle" with the provided 'make' and 'model' arguments

- without explicitly putting this, Java would implicitly attempt to call the superclass's default which would be the no-arg constructor
- Now if in the Car file, you didn't call super(...), then since there's no default super() constructor in base class, there would be a compilation error

→ What type of objects can you create?

- you can directly create objects of a subclass
  → `Car car = new Car();`
- objects of a superclass
  → `Vehicle vehicle = new Vehicle();`
- Objects with polymorphism — you can refer to them using a reference of the superclass type.
  → `Vehicle car = new Car();` → could then do `car.drive();`

→ Polymorphism (in inheritance)

- refers to the ability of objects of different subclasses to be treated as objects of their common superclass
- Two types
  - Overloading → compile-time
  - Overriding → run-time

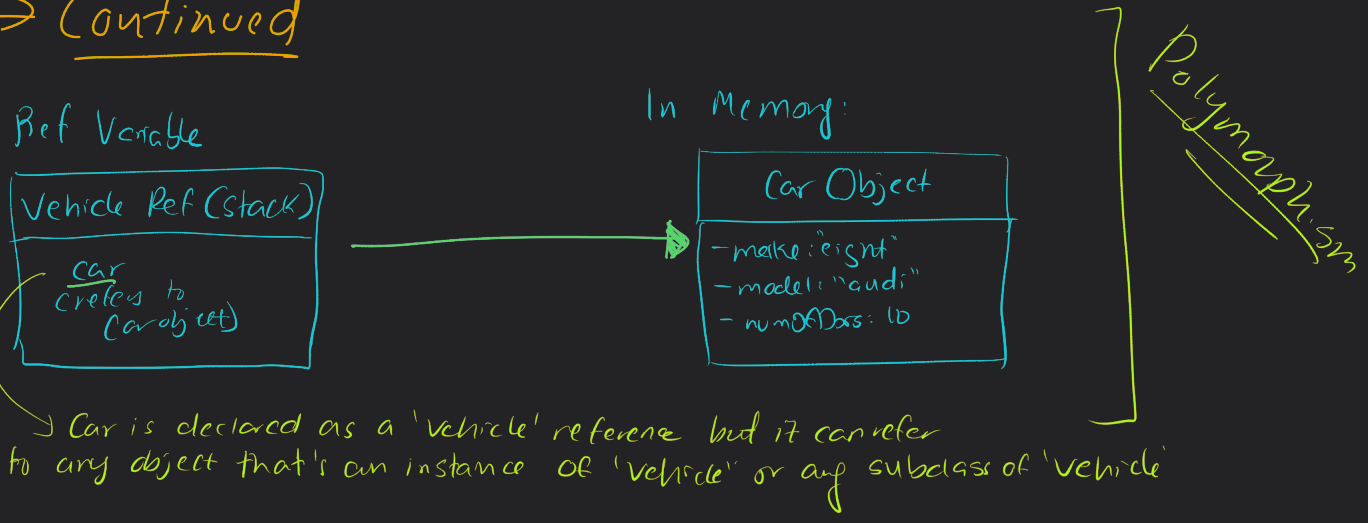
```
* Vehicle car = new Car("...", "...", 10);
car.start();
```

* Object is still of type Car but just referring to it using a reference of its superclass 'Vehicle' *

→ will be the one in car file

→ since car is type 'Vehicle' here, compiler checks for existence of 'start()' method in 'Vehicle' class which it finds
→ But, car points to an object of type Car due to polymorphism. JVM dynamically determines the actual type of the object ('Car') & looks for the implementation of start() in 'Car' class when 'start()' is invoked.

→ Continued



↳ Car is declared as a 'Vehicle' reference but it can refer to any object that's an instance of 'Vehicle' or any subclass of 'Vehicle'

→ Allowed & Not Allowed

- If a variable is of type Mammal, you can store a reference to any object that "is a" Mammal.

```
Mammal md = new Dolphin() // OK
Dolphin flipper = new Mammal() // Not OK
```

↳ in Vehicle example

```
Vehicle car = new Car(); // OK
Car car = new Vehicle() // NOT OK
```

↳ Vehicle is not necessarily a car so can't assign an instance of a superclass to a reference variable of its subclass type cuz superclass won't have all attributes/meth that are defined in subclass

- If a method is expecting a Mammal object, the argument can be any object that "is a" Mammal.

```
Public static void doThis(Mammal x) {...}
```

assumes Dog extends...

→ could do doThis(dog)
↳ Dog dog = new Dog();
↳ could have
Mammal dog = new Dog();

- If a variable is of type Vehicle, then you can only call methods of the Vehicle class.

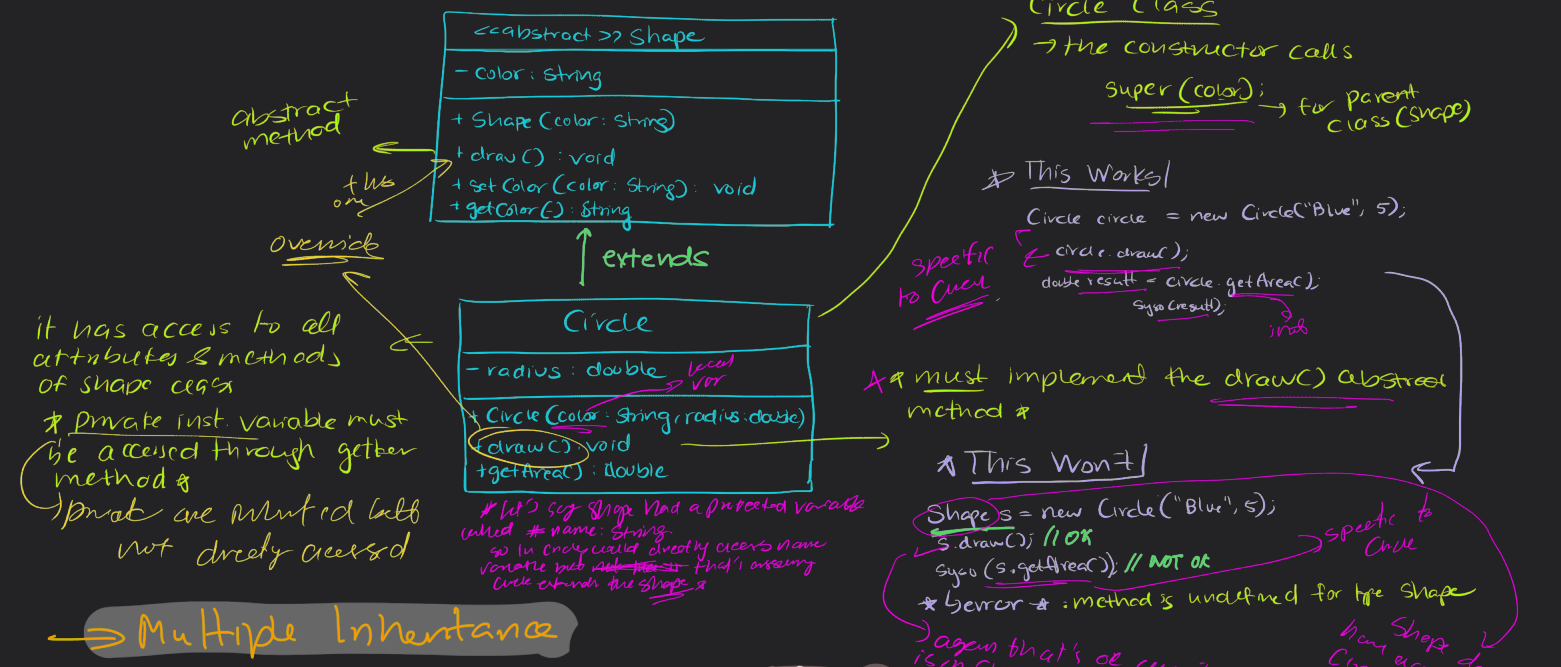
```
Vehicle car = new Car(...);
car.startC(); // OK
car.drive(); // NOT OKAY
```

↳ specific to Car only

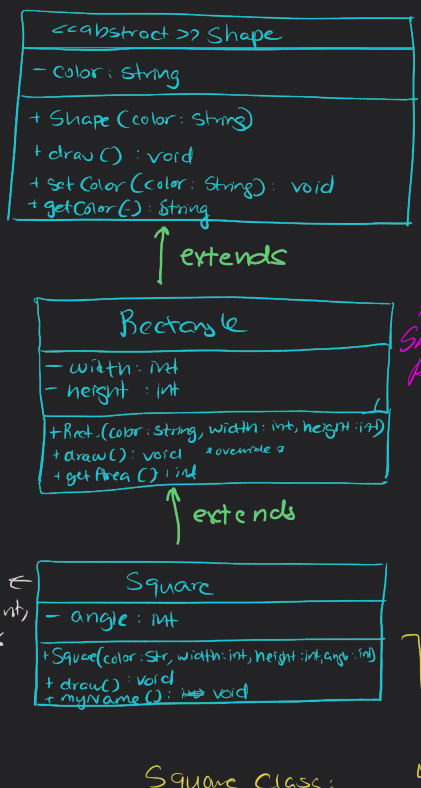
could cast it → ((Car)car).drive();

→ Abstract classes & Inheritance

- Abstract classes:
  - declared using abstract keyword
  - can have abstract methods (without a body) as well as concrete methods
  - CAN'T be instantiated directly like no object creation of abstract class
  - Any class that extends it MUST implement all its abstract methods unless the subclass itself is abstract



→ Multiple Inheritance



→ Error

- Square s = new Square(...);
  - s.draw(); s.getColor(); & object class methods
  - s.getArea() s.setColor()
  - s.myName(); → would work

Basically everything available from Shape → Square

- Rectangle s = new Square(...);
  - s.draw();
  - s.getArea(); & object stuff
  - s.get(color/setColor)
  - s.myName(); → wouldn't work

from Shape → Rectangle

NO

- Shape s = new Square(...);
  - s.draw();
  - s.getArea(); // NOT HAVE
  - s.myName(); // NOT

s.getColor();
s.getDim();
& object class

Square Class:

```
private int angle;
public Square(....) {                    → String color, int width, int height
    super(color, width, height);         → first line *
    this.angle = angle;
}
@Override
public void draw() {
    sysout(... + getColor() ... angle)
}
```

Even if Square class didn't have draw, you could still call draw method because it's in the parent class which is Rectangle

since color is a private variable you would need to do getColor()

@Override

```
public class Person {
    Private String name;
    public Person(String name) {
        this.name = name;
    }
    Public String toString() {
        return "Person [name=" + name + "]";
    }
}
```

1) Omit the @override, then recompile
→ works ✓

2) Omit the @override & misspell the toString method name & recompile
→ if we misspell it, it will still compile, cuz program will think toString() is a new method

3) Include the @Override & misspell the toString() method name & recompile

so like

```
@Override
public cString toString() {
    return "Person [name=" + name + "]";
}
```

→ compile error because the compiler sees the @Override annotation & checks for a method in the superclass that matches the signature of an old method. If it doesn't find one, it throws a compile error.

Method Signature

add(int, int)
↳ includes method name & parameter list
NOT return type or access modifier