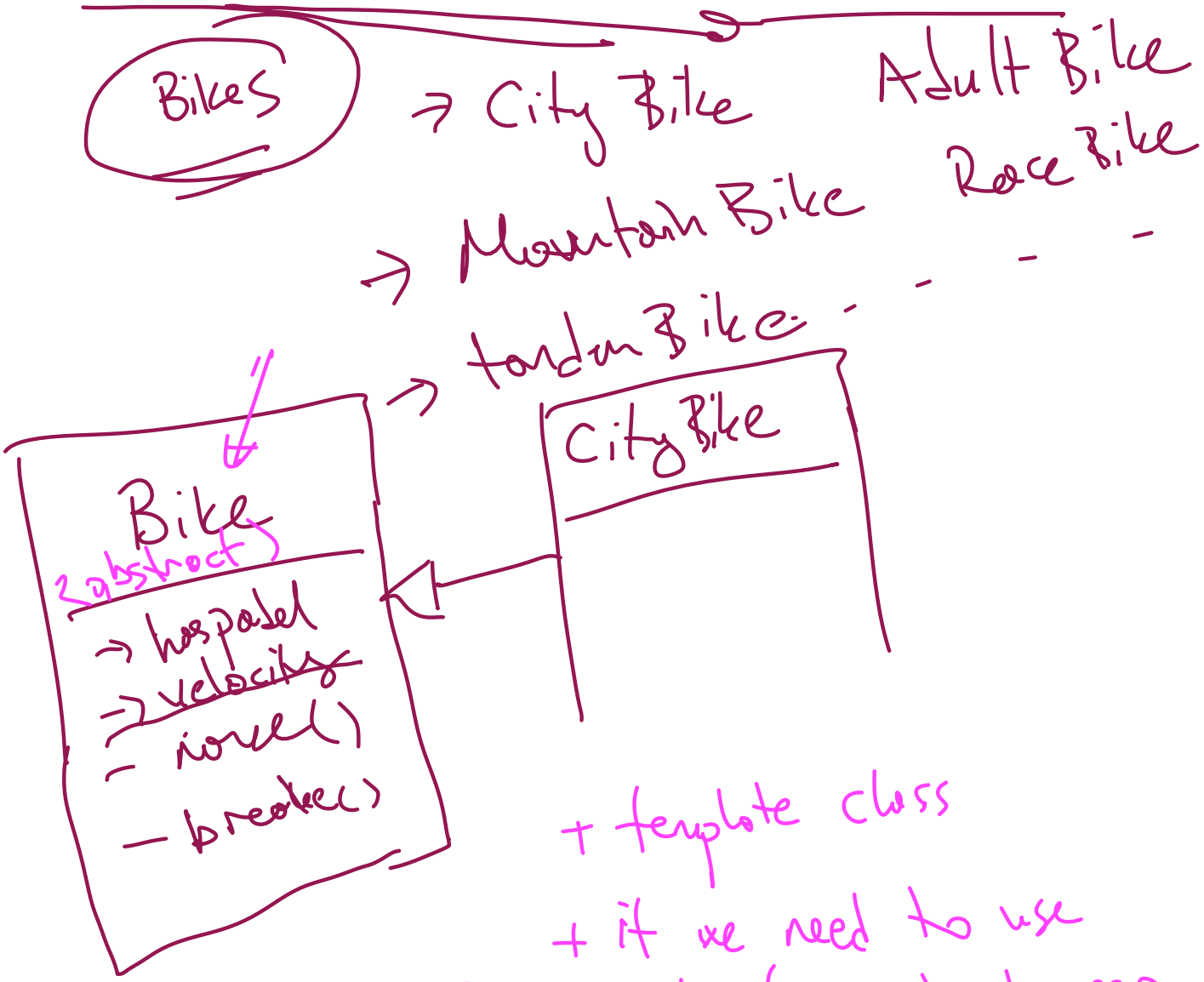
- DustBin
- N.Light
- elevator

labs. I will try to implement

---

inheritance
encapsulation
polymorphism

abstract class



+ template class
+ if we need to use same interface between

Subclasses -
+ if we use abstract, we are going to have more loosly coupled desing.

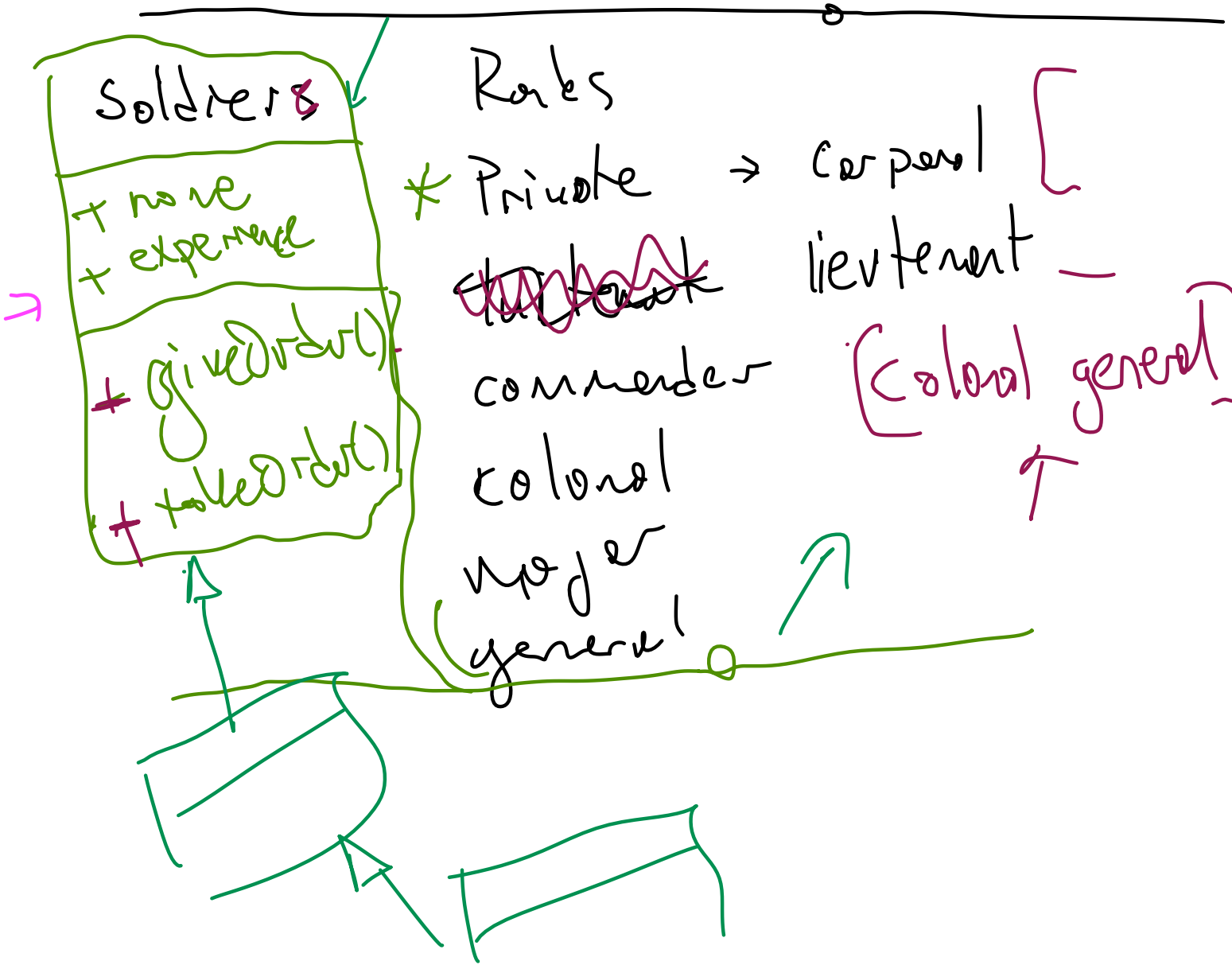
LC ∞ design:
it will be easier for us to extend our code without modification!

→ Open-Close principle of Desing

→ Rule of thump you should always (best practices)

abstract classes / or interfaces for

Method returns, arguments, fields



public abstract class Soldier{

private int experience;
private String name;

public abstract giveOrder();
public abstract takeOrder();

}

class Battalion {

Soldier[] crew = new Soldier[1000];

Soldier commander;

}

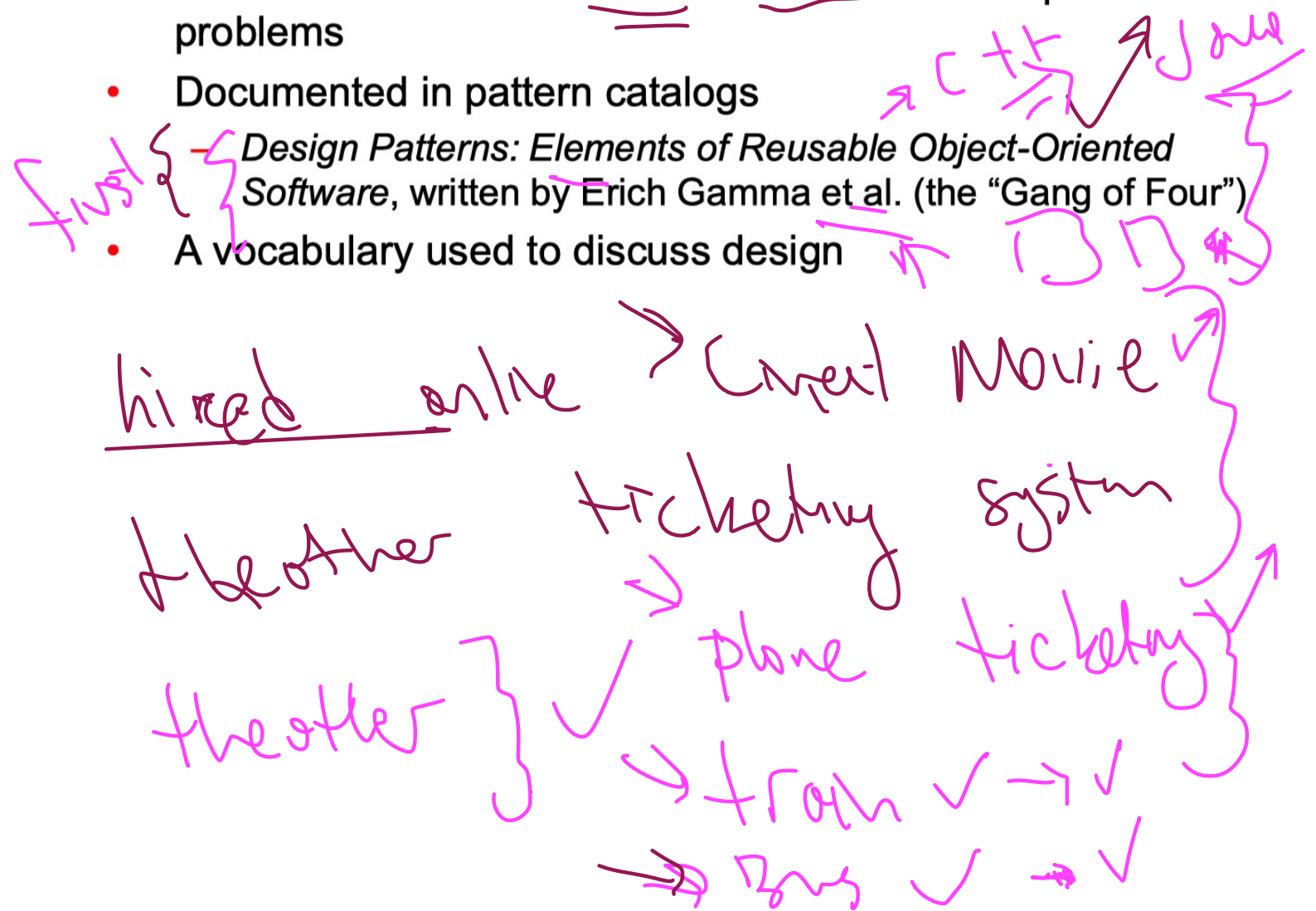
Soldier s = new Private();

Soldier s2 = new Commander();

Design Patterns

Design patterns are:

- Reusable solutions to common software development problems
- Documented in pattern catalogs
  - *Design Patterns: Elements of Reusable Object-Oriented Software*, written by Erich Gamma et al. (the "Gang of Four")
- A vocabulary used to discuss design



Singleton Pattern

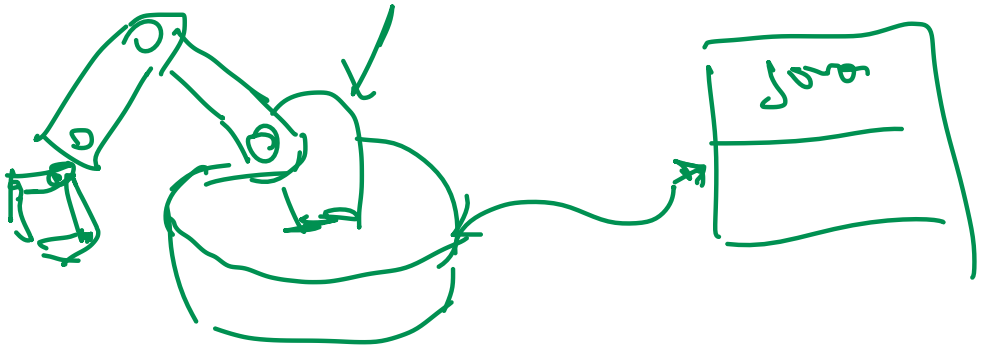
The singleton design pattern details a class implementation that can be instantiated only once.

```
public class SingletonClass {
1   private static final SingletonClass instance =
        new SingletonClass();

2   private SingletonClass() {}

    public static SingletonClass getInstance() {
3       return instance;
    }
}
```

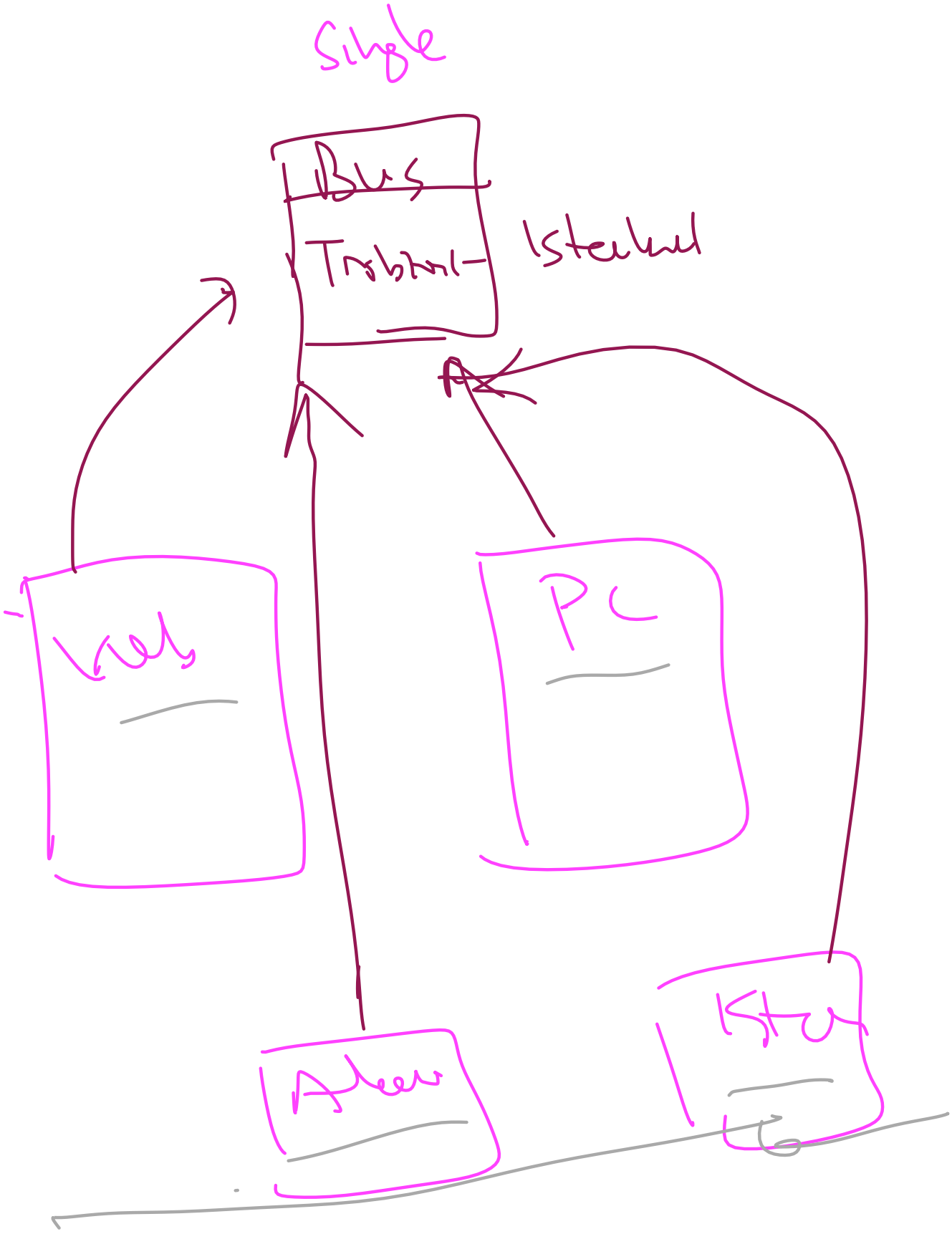
private Constructor

Session 4

SingletonClass c = new SingletonClass();



SingletonClass.getInstance();


Single
Bus
PC

Nested Classes

A nested class is a class declared within the body of another class. Nested classes:

- Have multiple categories
  - Inner classes
    - Member classes
    - Local classes
    - Anonymous classes
  - Static nested classes
- Are commonly used in applications with GUI elements
- Can limit utilization of a "helper class" to the enclosing top-level class

```
public class A {
    public class B {
    }// end of B
}// end of A
A.B ab = new A.B();
```

Inner Class: Example

```
public class Car {
    private boolean running = false;
    private Engine engine = new Engine();

    private class Engine {
        public void start() {
            running = true;
        }
    }

    public void start() {
        engine.start();
    }
}
```

Anonymous Inner Classes

An anonymous class is used to define a class with no name.

```
public class AnonymousExampleClass {
    public Object o = new Object() {
        @Override
        public String toString() {
            return "In an anonymous class method";
        }
    };
}
```

Car c = new Car();

Inheritance with Java Interfaces

6

- principle oo designs

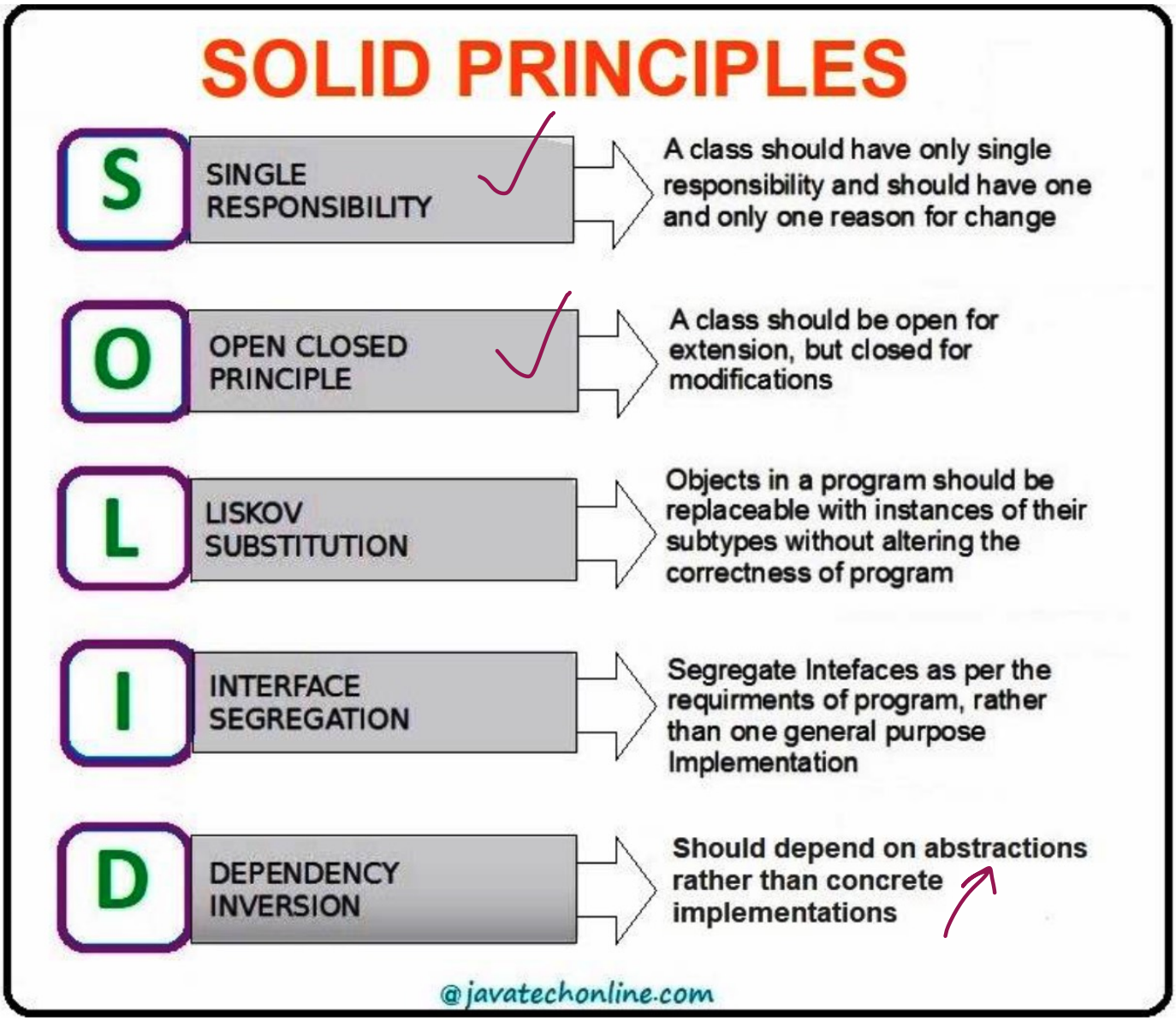
S → Single purpose

O → open - close principle

L → Liskov

I →

D → dependency inversion

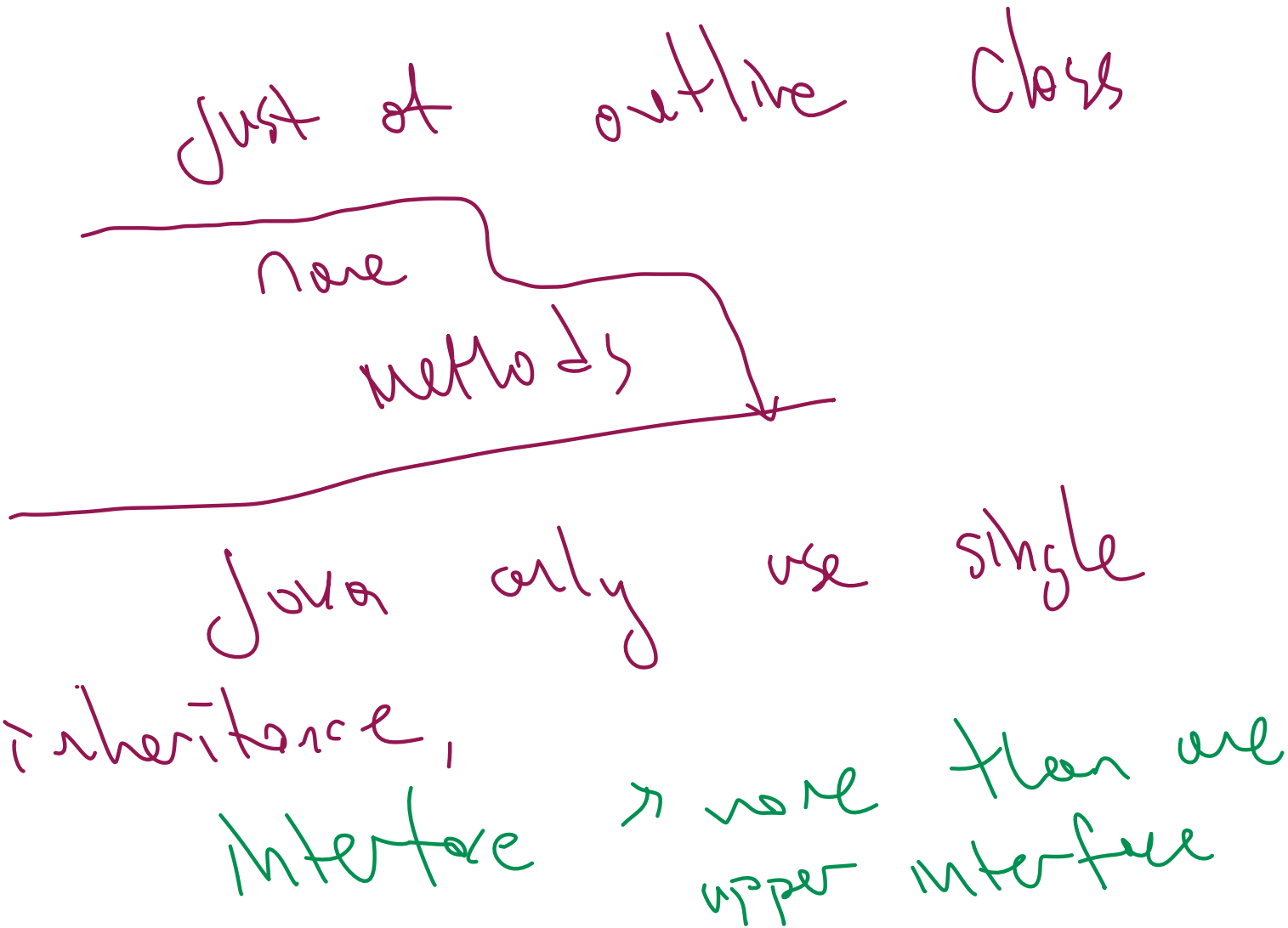


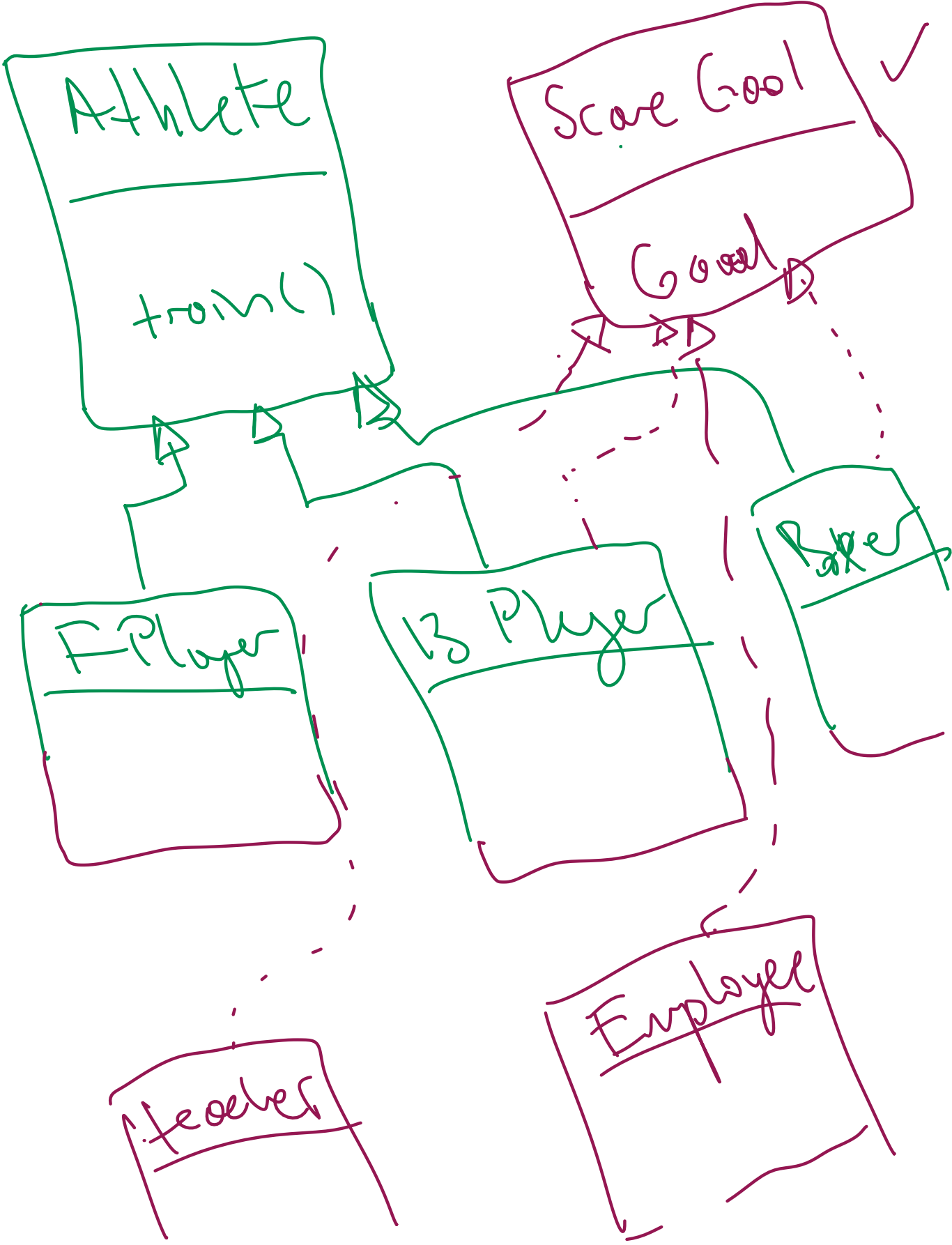
Design pattern →

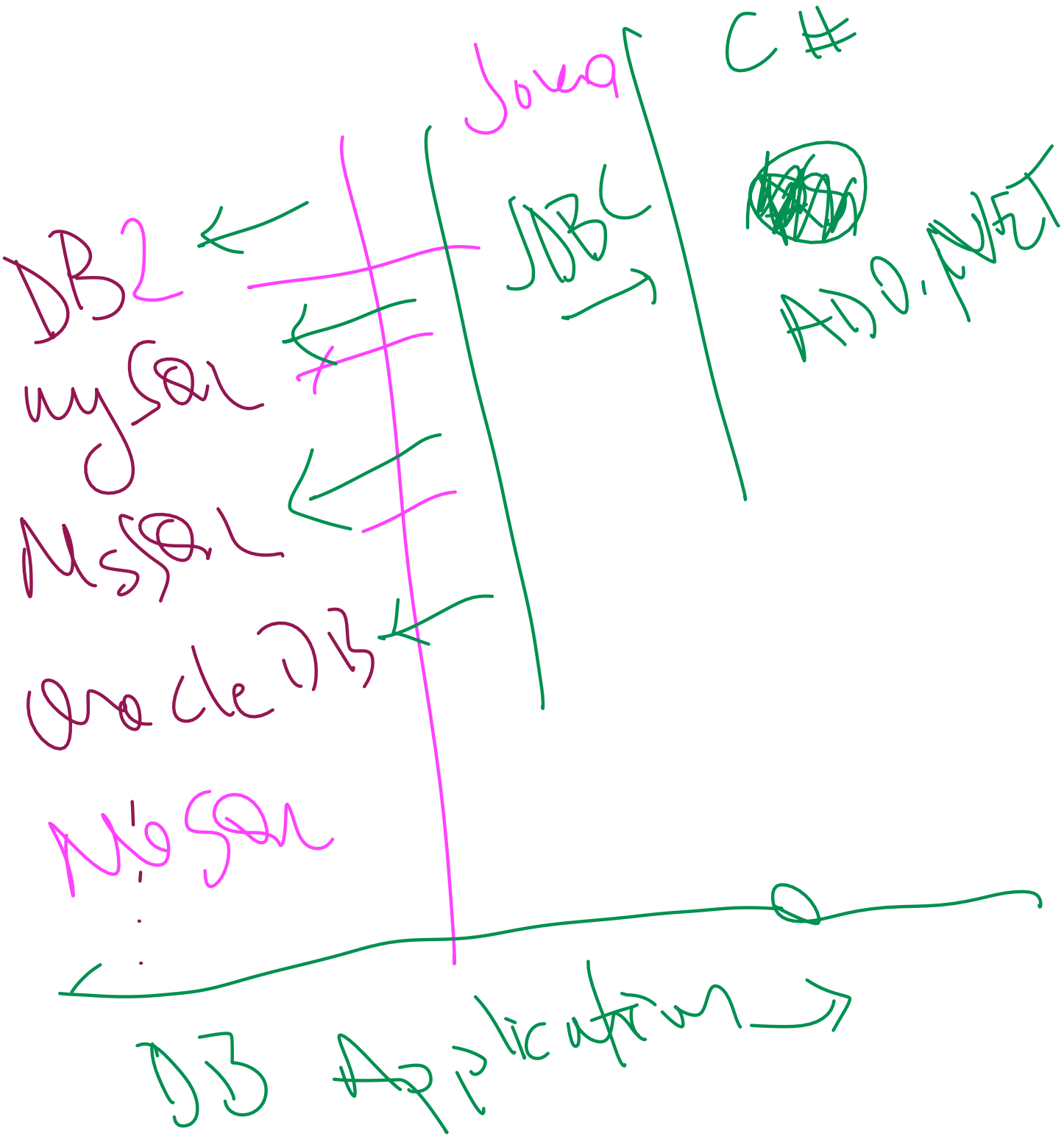
Implementation Substitution

The ability to outline abstract types is a powerful feature of Java. Abstraction enables:

- Ease of maintenance
  - Classes with logic errors can be substituted with new and improved classes.
- Implementation substitution
  - The `java.sql` package outlines the methods used by developers to communicate with databases, but the implementation is vendor-specific.
- Division of labor
  - Outlining the business API needed by an application's UI allows the UI and the business logic to be developed in tandem.



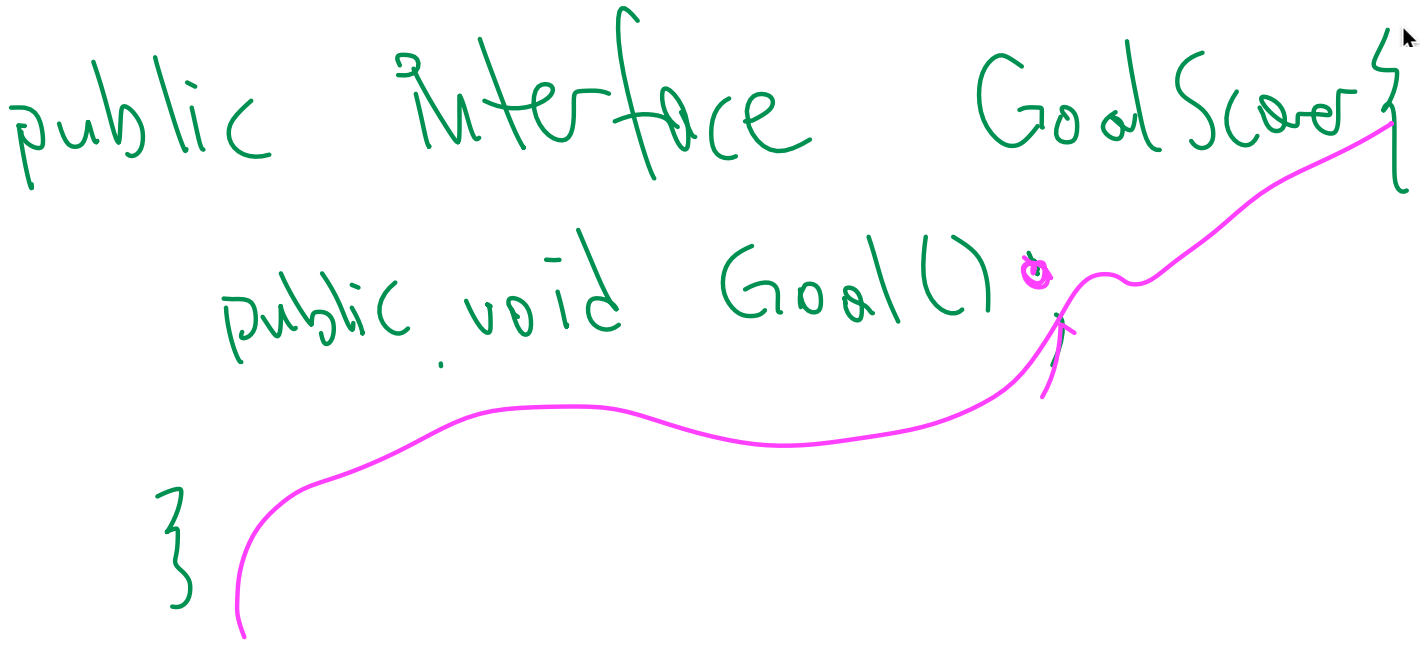

Athlete
train()
Score Goal
Goal
FPlayer
LB Player
Baker
Employee
Teacher


Java
C#
JDBC
ADO.NET
DB2
MySQL
MSSQL
Oracle
NoSQL
DB Application

Java Interfaces

Java interfaces are used to define abstract types. Interfaces:

- Are similar to abstract classes containing only public abstract methods
- Outline methods that must be implemented by a class
  - Methods must not have an implementation {braces}.
- Can contain constant fields
- Can be used as a reference type
- Are an essential component of many design patterns

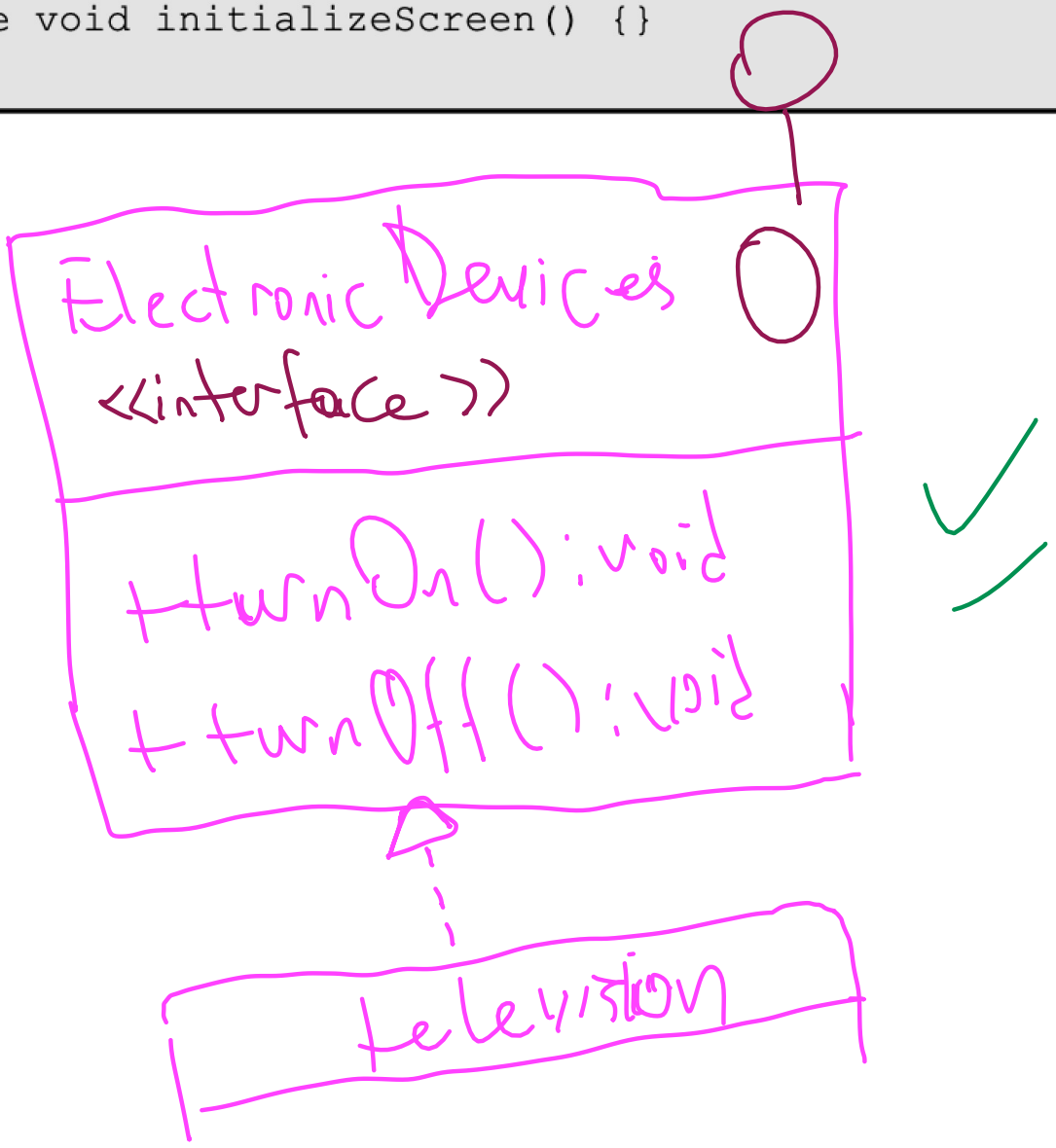


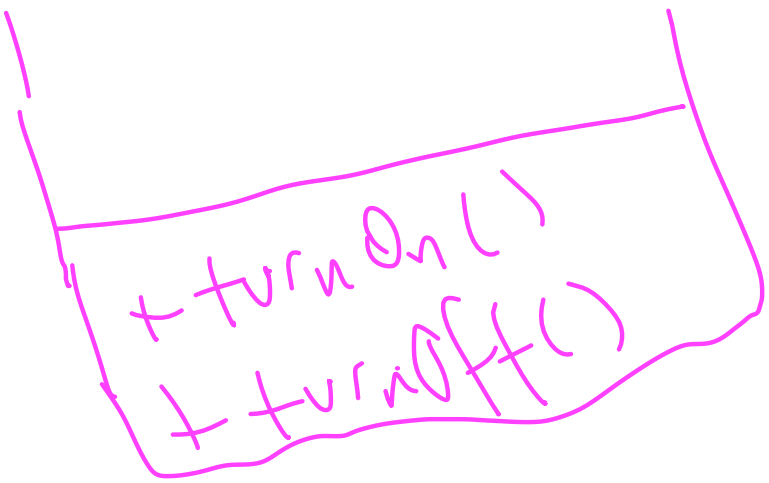
Developing Java Interfaces

Public, top-level interfaces are declared in their own `.java` file.
You implement interfaces instead of extending them.

```
public interface ElectronicDevice {
    public void turnOn();
    public void turnOff();
}
```

```
public class Television implements ElectronicDevice {
    public void turnOn() { }
    public void turnOff() { }
    public void changeChannel(int channel) {}
    private void initializeScreen() {}
}
```





Television tv = new Television();

ElectronicDevice ed = new Television();

ed.turnOn()

ed.turnOff();

anonymous

ElectricDevice ed = new ElectronicDevice(){

public void turnOn(){..}

public void turnOff(){}}

Constant Fields

Interfaces can have constant fields.

```
public interface ElectronicDevice {
    public static final String WARNING =
        "Do not open, shock hazard";
    public void turnOn();
    public void turnOff();
}
```

Interface References

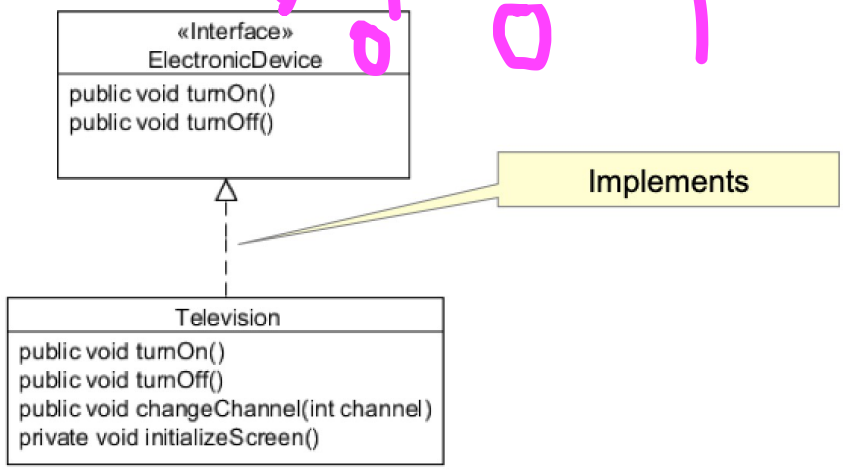
You can use an interface as a reference type. When using an interface reference type, you must use only the methods outlined in the interface.

```
ElectronicDevice ed = new Television();
ed.turnOn();
ed.turnOff();
ed.changeChannel(2); // fails to compile
String s = ed.toString();
```

`instanceof` Operator

You can use `instanceof` with interfaces.

```
Television t = new Television();
if (t instanceof ElectronicDevice) { }
```



Television is an instance of an ElectronicDevice.

Marker Interfaces

- Marker interfaces define a type but do not outline any methods that must be implemented by a class.

```
public class Person implements java.io.Serializable { }
```

- The only reason these type of interfaces exist is type checking.

```
Person p = new Person();
if (p instanceof Serializable) {

}
```

Casting to Interface Types

You can cast to an interface type.

```
public static void turnObjectOn(Object o) {
    if (o instanceof ElectronicDevice) {
        ElectronicDevice e = (ElectronicDevice)o;
        e.turnOn();
    }
}
```

Using Generic Reference Types

- Use the most generic type of reference wherever possible:

```
EmployeeDAO dao = new EmployeeDAOMemoryImpl();
dao.delete(1);
```

`EmployeeDAOMemoryImpl` implements `EmployeeDAO`

- By using an interface reference type, you can use a different implementing class without running the risk of breaking subsequent lines of code:

```
EmployeeDAOMemoryImpl dao = new EmployeeDAOMemoryImpl();
dao.delete(1);
```

It is possible that you could be using `EmployeeDAOMemoryImpl` only methods here.

Implementing and Extending

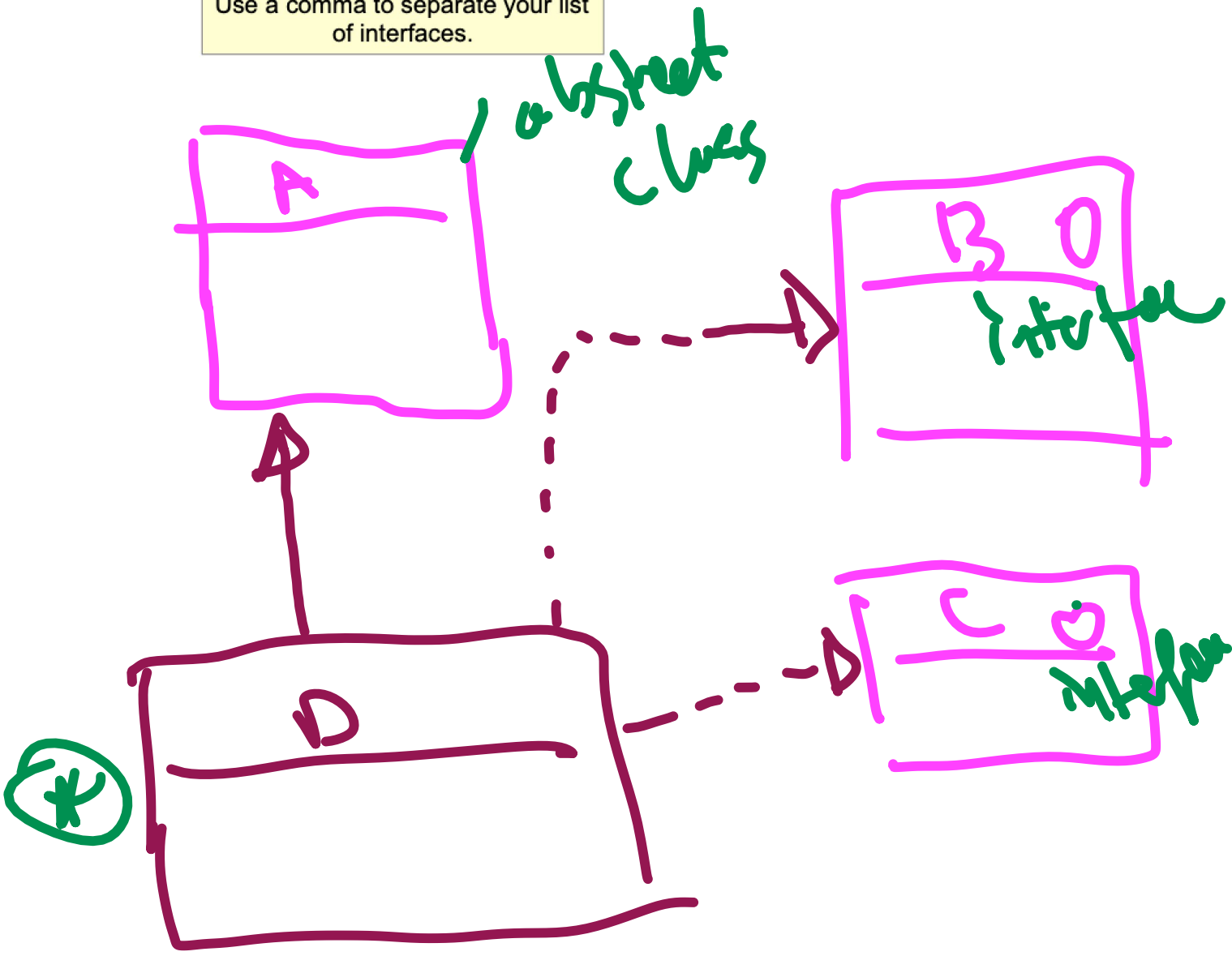
- Classes can extend a parent class and implement an interface:

```
public class AmphibiousCar extends BasicCar implements
MotorizedBoat { }
```

- You can also implement multiple interfaces:

```
public class AmphibiousCar extends BasicCar implements
MotorizedBoat, java.io.Serializable { }
```

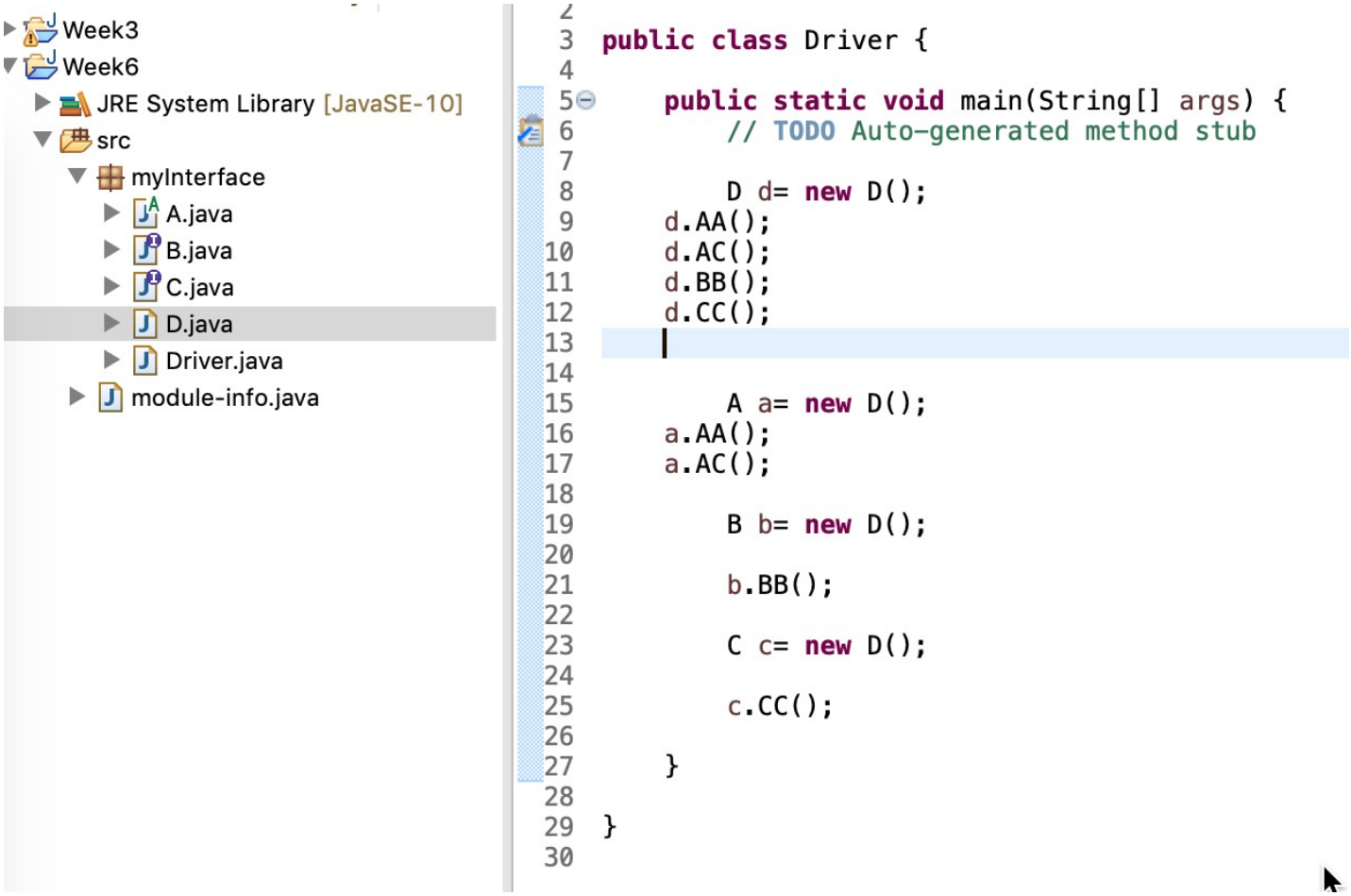
Use a comma to separate your list of interfaces.

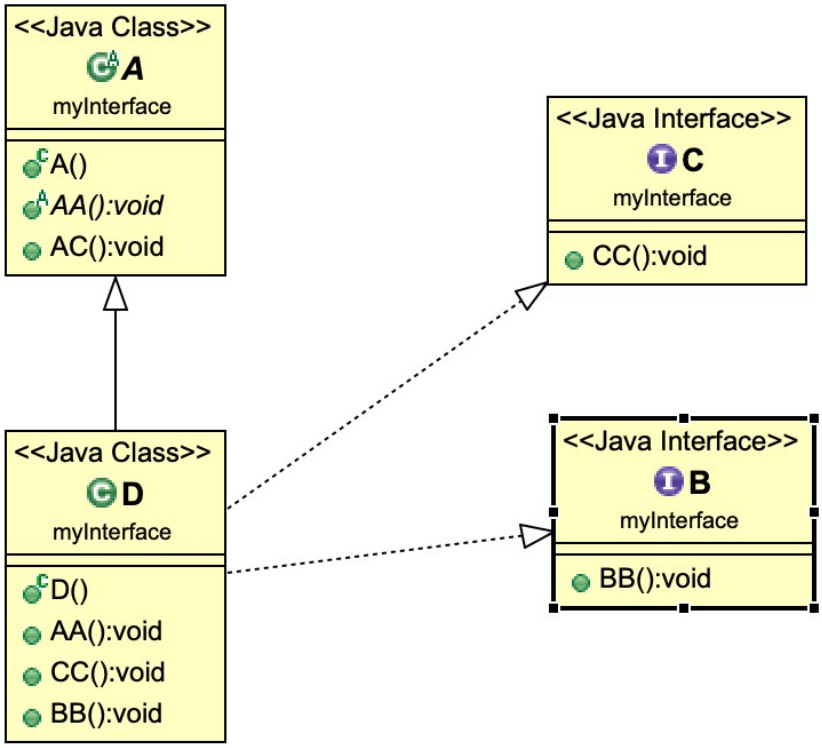


⊛public class D extends A implements B, C {

A a= new D();

B b= new D();





Extending Interfaces

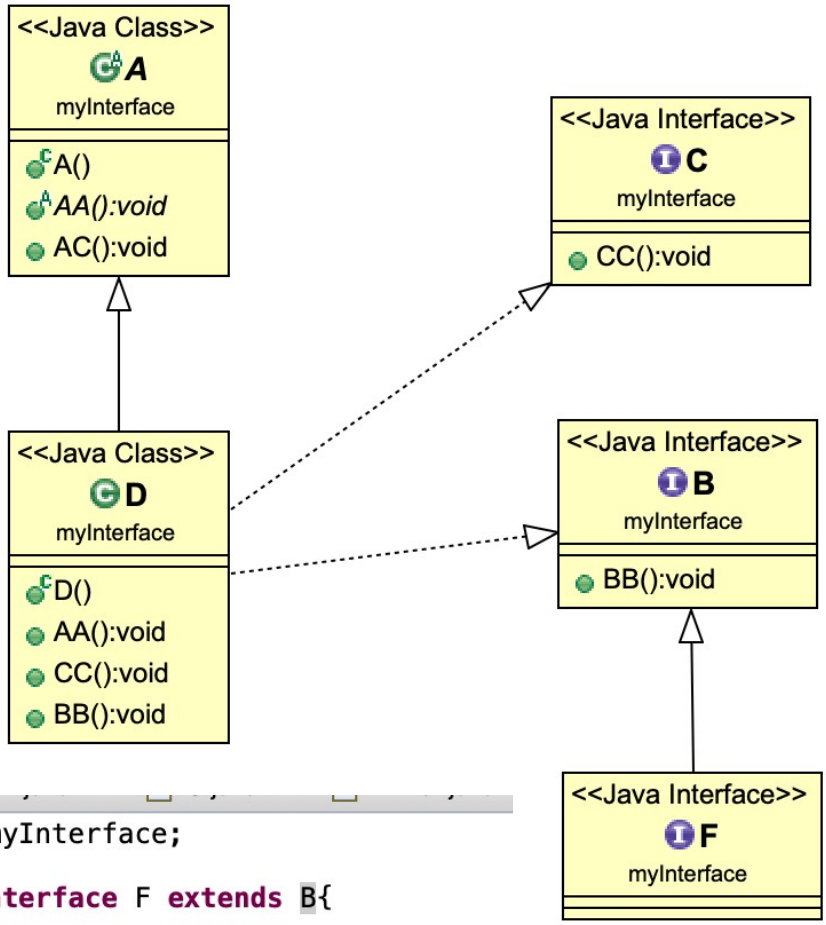
- Interfaces can extend interfaces:

```
public interface Boat { }
```

```
public interface MotorizedBoat extends Boat { }
```

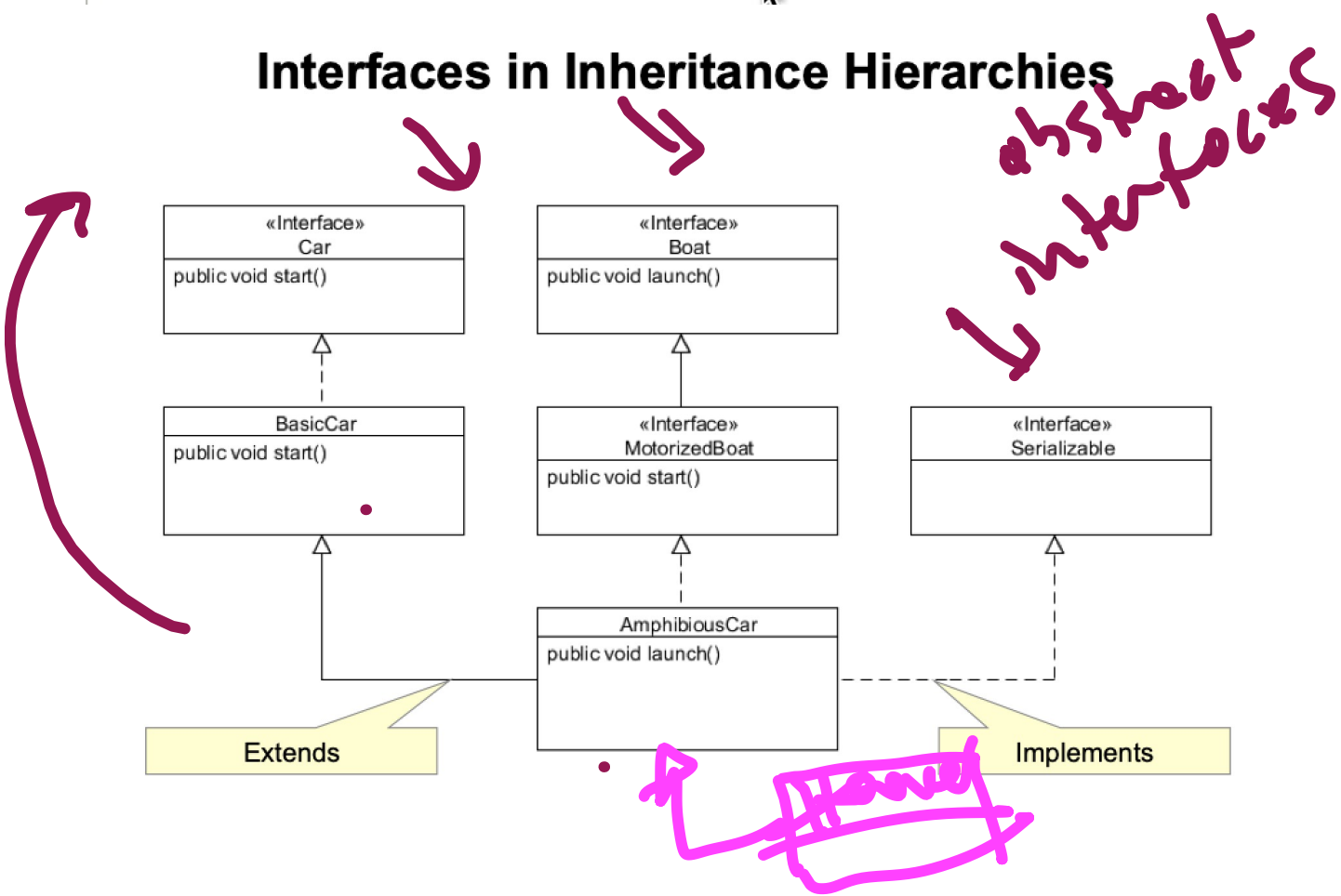
- By implementing `MotorizedBoat`, the `AmphibiousCar` class must fulfill the contract outlined by both `MotorizedBoat` and `Boat`:

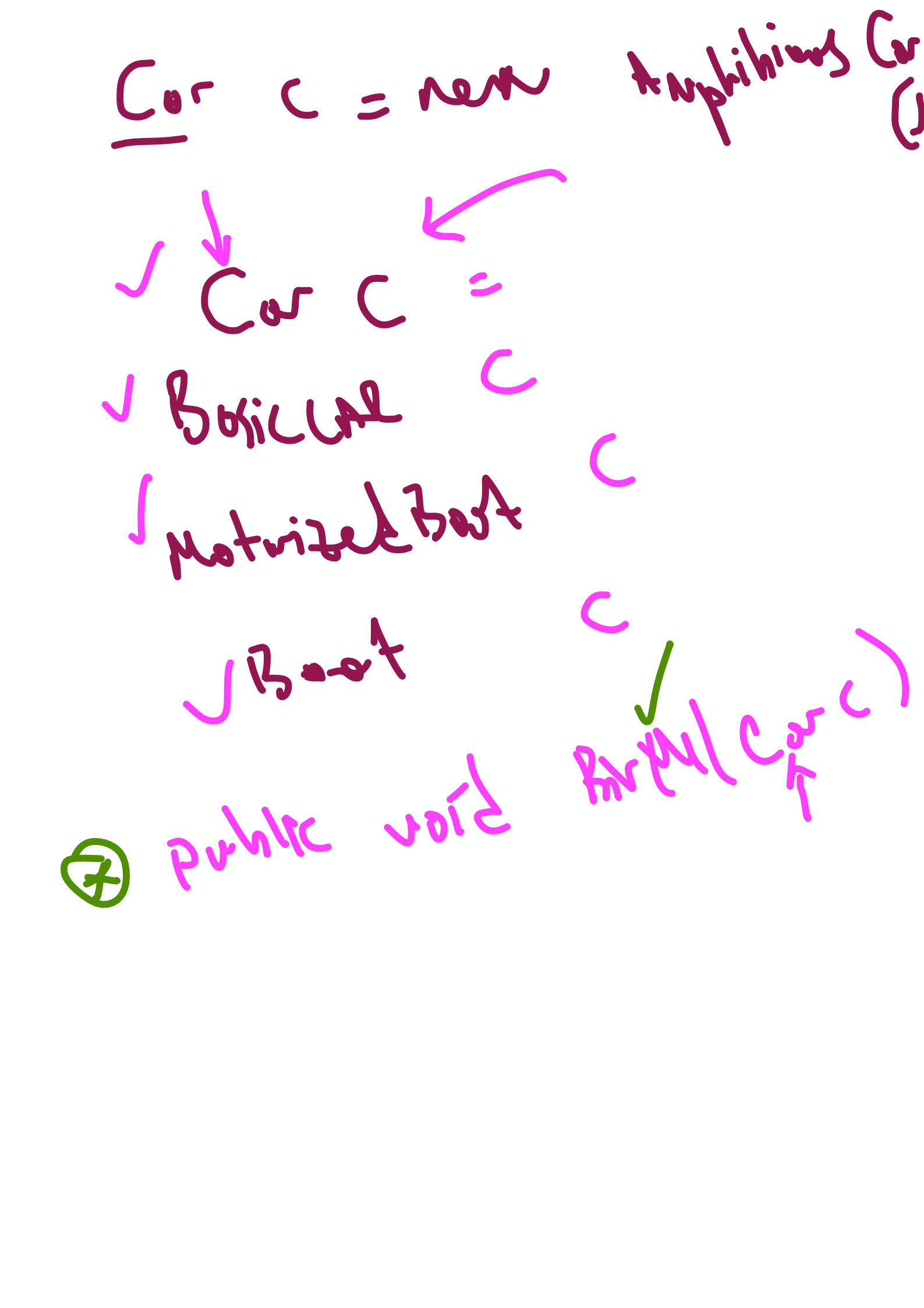
```
public class AmphibiousCar extends BasicCar implements
MotorizedBoat, java.io.Serializable { }
```




```
1  package myInterface;
2
3  public interface F extends B{
4
5  }
6
```

Interfaces in Inheritance Hierarchies




Car c = new
Amphibious Car
(
Car c
Basic Car
Motorized Boat
Boat
c
c
c
c
public void
Run(Car c)

Design Patterns and Interfaces

- One of the principles of object-oriented design is to:
  "Program to an interface, not an implementation."
- This is a common theme in many design patterns. This principle plays a role in:
  - The DAO design pattern ✓
  - The Factory design pattern ✓

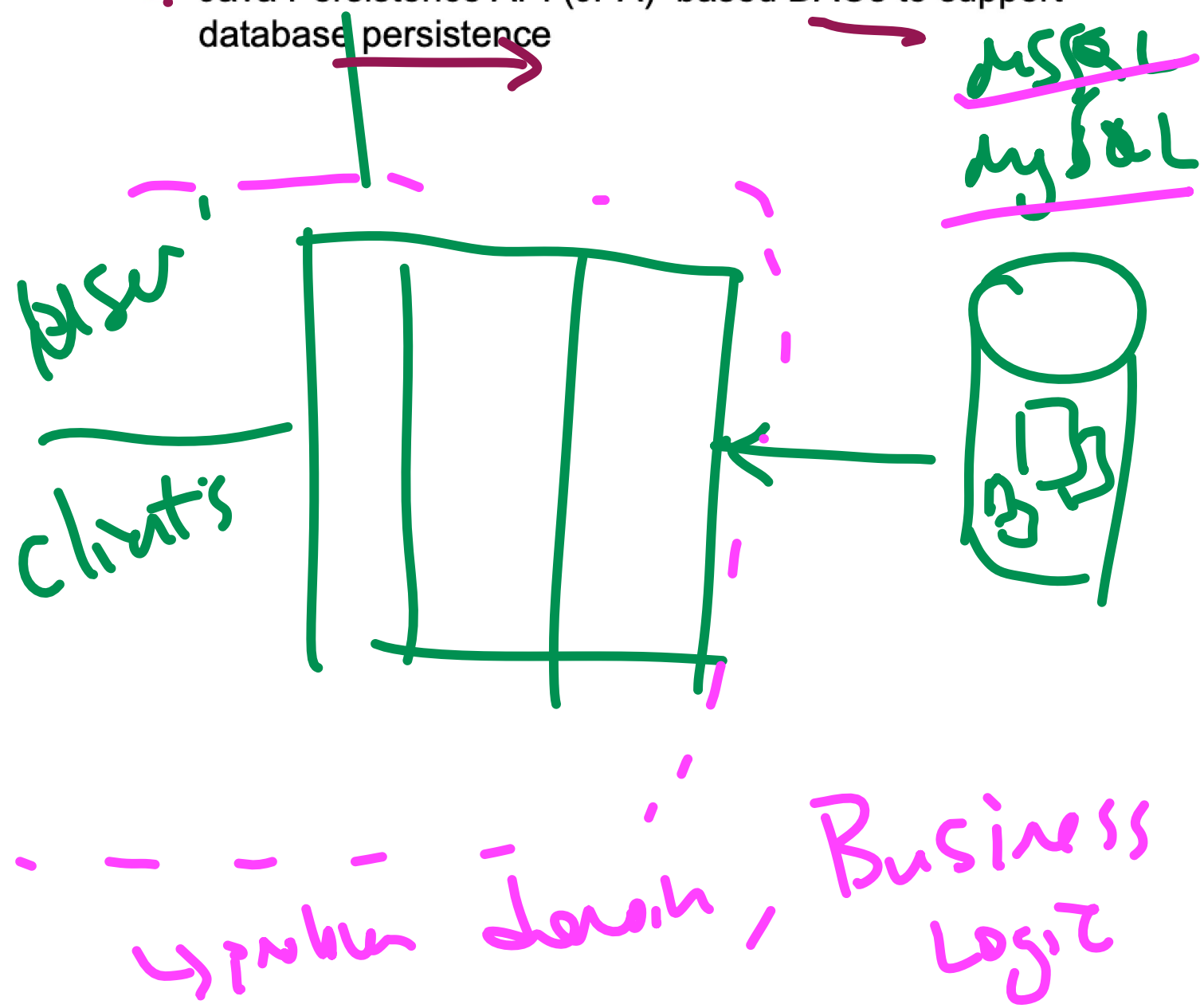
⇒

use abstractions (abstract classes & interfaces) to program in the field of Classes in return types of methods in arguments of Methods

DAO Pattern

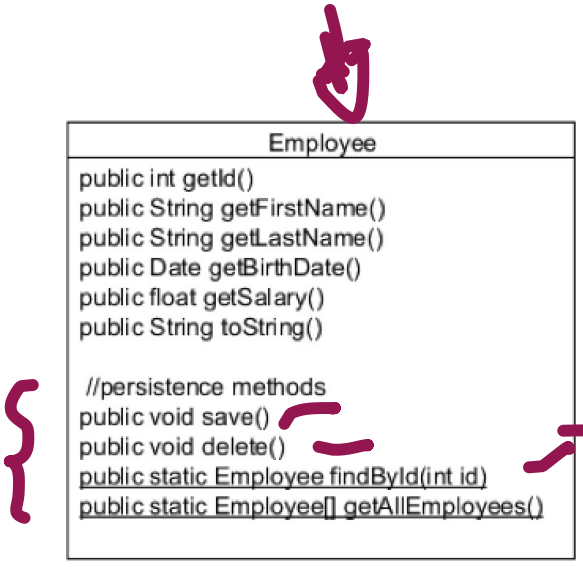
The Data Access Object (DAO) pattern is used when creating an application that must persist information. The DAO pattern:

- Separates the problem domain from the persistence mechanism
- Uses an interface to define the methods used for persistence. An interface allows the persistence implementation to be replaced with:
  - – Memory-based DAOs as a temporary solution
  - – File-based DAOs for an initial release
  - – JDBC-based DAOs to support database persistence
  - – Java Persistence API (JPA)–based DAOs to support database persistence



Before the DAO Pattern

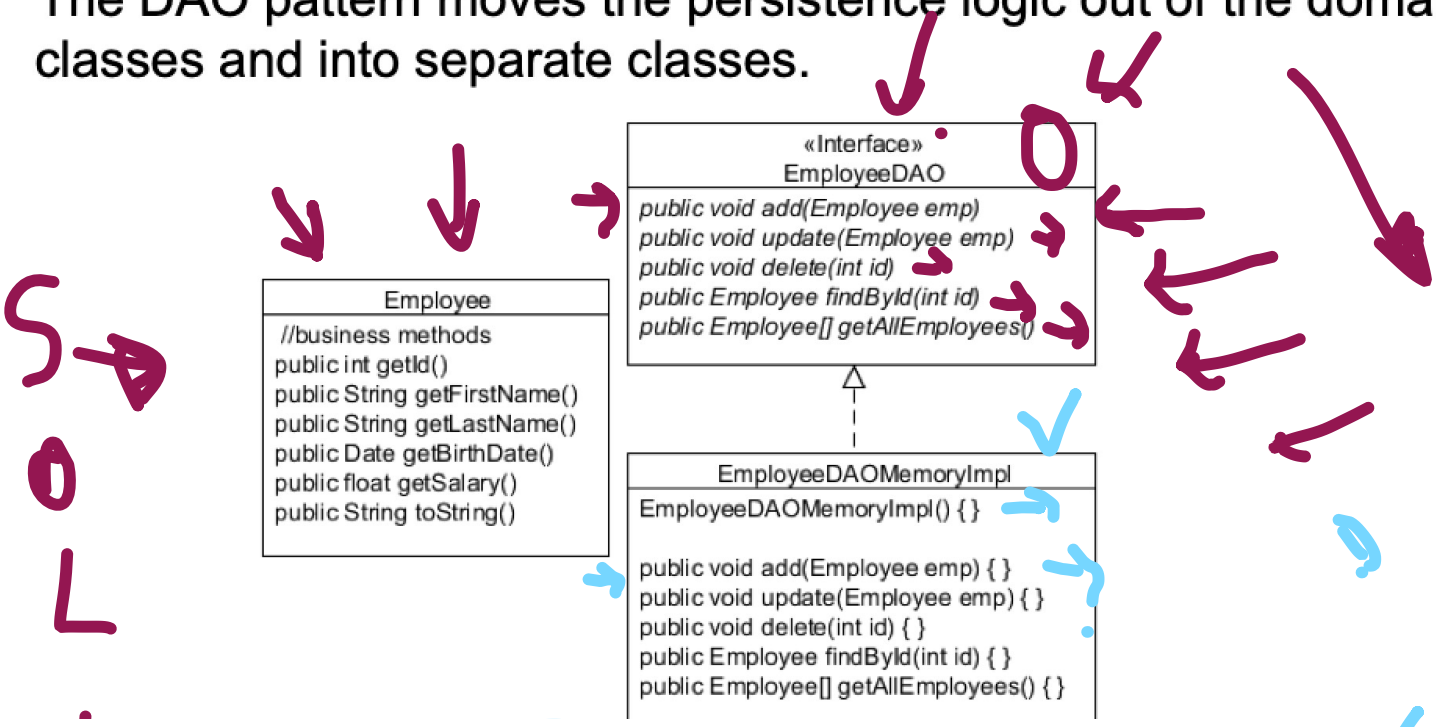
Notice the persistence methods mixed in with the business methods.



Before the DAO pattern

After the DAO Pattern

The DAO pattern moves the persistence logic out of the domain classes and into separate classes.



The Need for the Factory Pattern

The DAO pattern depends on using interfaces to define an abstraction. Using a DAO implementation's constructor ties you to a specific implementation.

```
EmployeeDAO dao = new EmployeeDAOMemoryImpl();
```

With use of an interface type, any subsequent lines are not tied to a single implementation.

This constructor invocation is tied to an implementation and will appear in many places throughout an application.

Using the Factory Pattern

Using a factory prevents your application from being tightly coupled to a specific DAO implementation.

```
EmployeeDAOFactory factory = new EmployeeDAOFactory();
EmployeeDAO dao = factory.createEmployeeDAO();
```

The `EmployeeDAO` implementation is hidden.

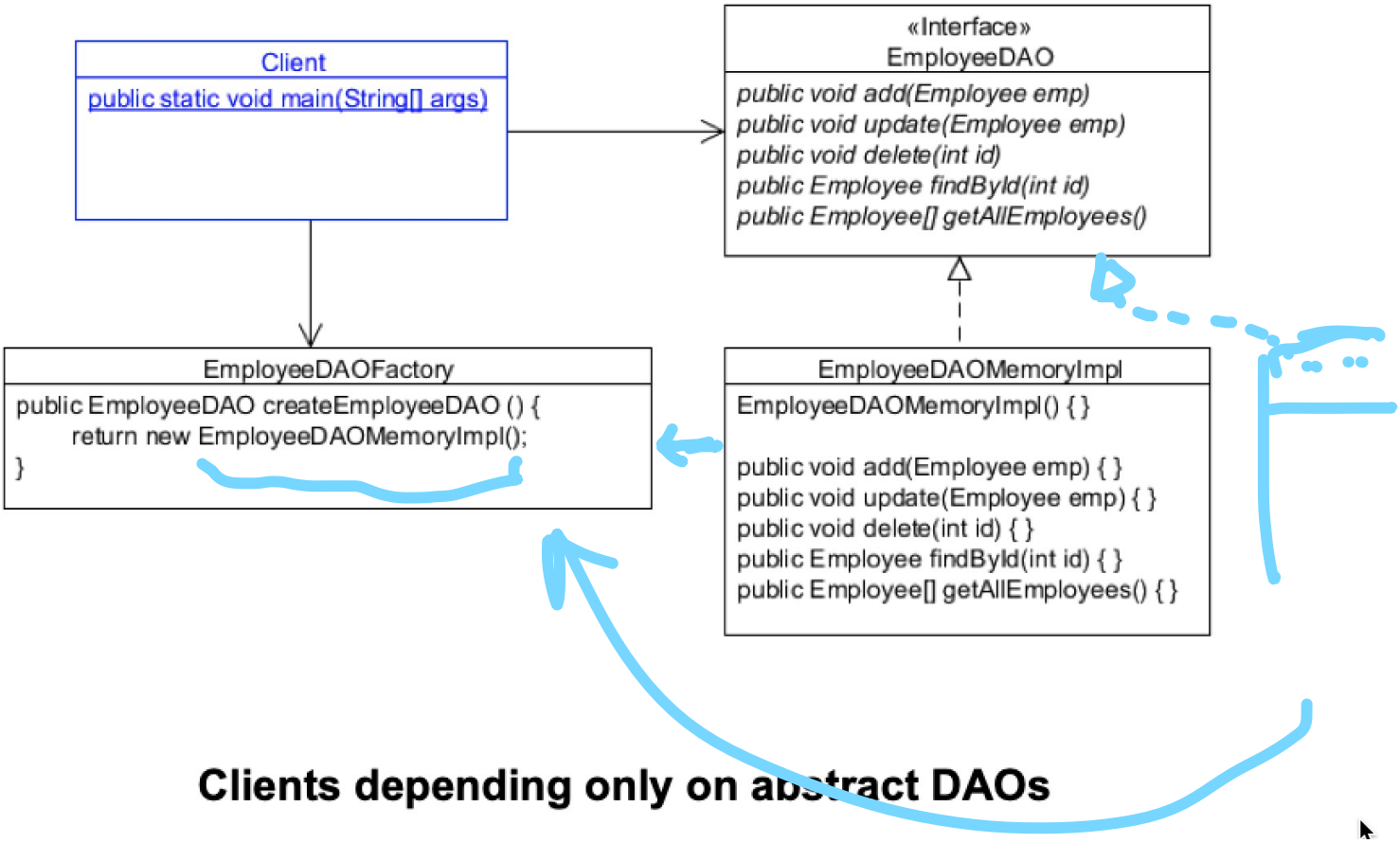
The Factory

The implementation of the factory is the only point in the application that should depend on concrete DAO classes.

```
public class EmployeeDAOFactory {

    public EmployeeDAO createEmployeeDAO() {
        return new EmployeeDAOMemoryImpl();
    }
}
```

Returns an interface typed reference

The DAO and Factory Together

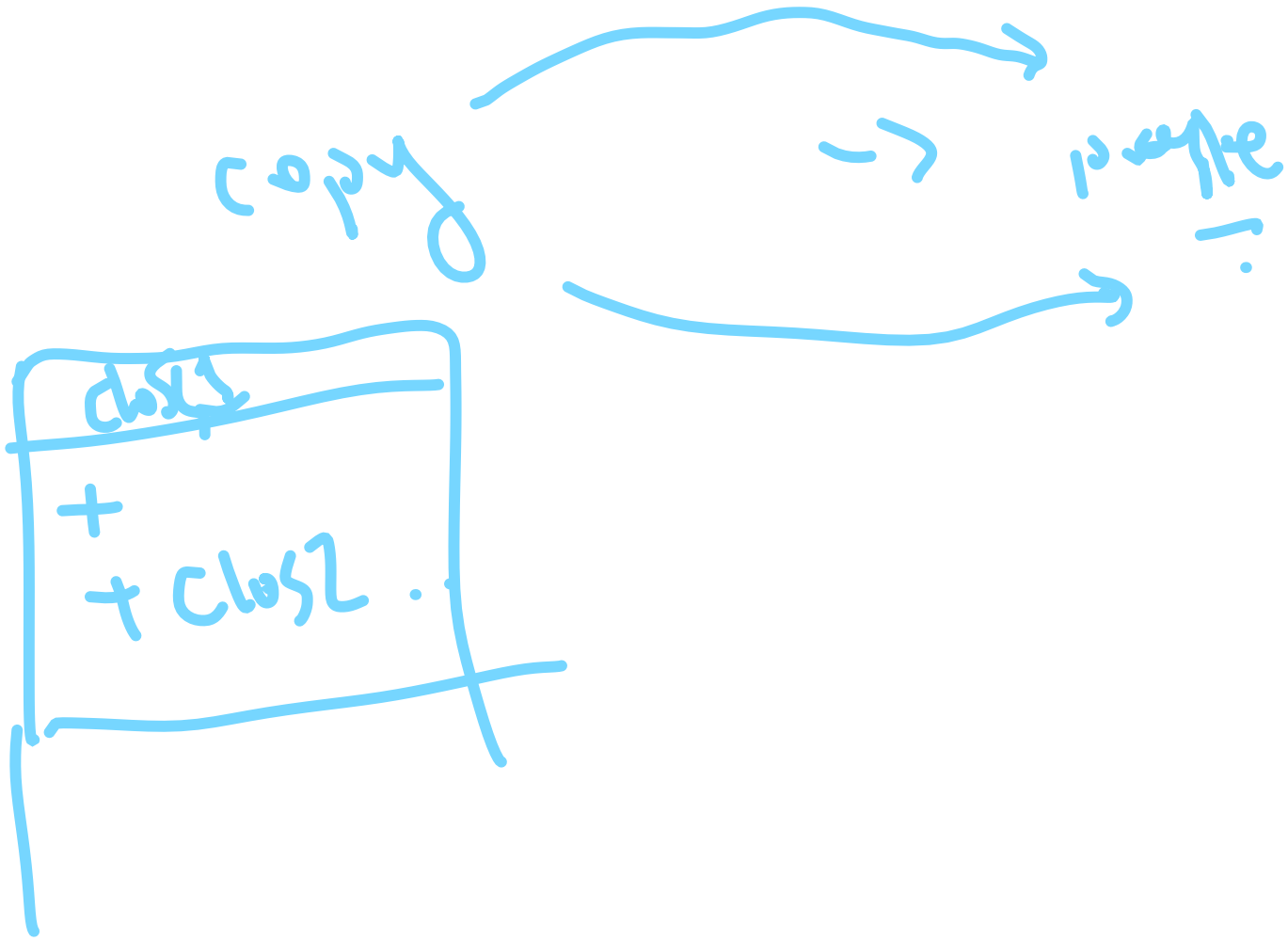


Clients depending only on abstract DAOs

Code Reuse

Code duplication (copy and paste) can lead to maintenance problems. You do not want to fix the same bug multiple times.

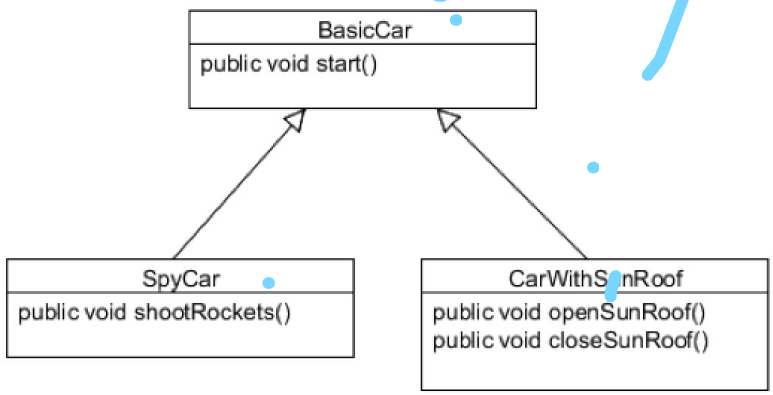
- "Don't repeat yourself!" (DRY principle)
- Reuse code in a good way:
  - Refactor commonly used routines into libraries.
  - Move the behavior shared by sibling classes into their parent class.
  - Create new combinations of behaviors by combining multiple types of objects together (composition).



Design Difficulties

Class inheritance allows for code reuse but is not very modular

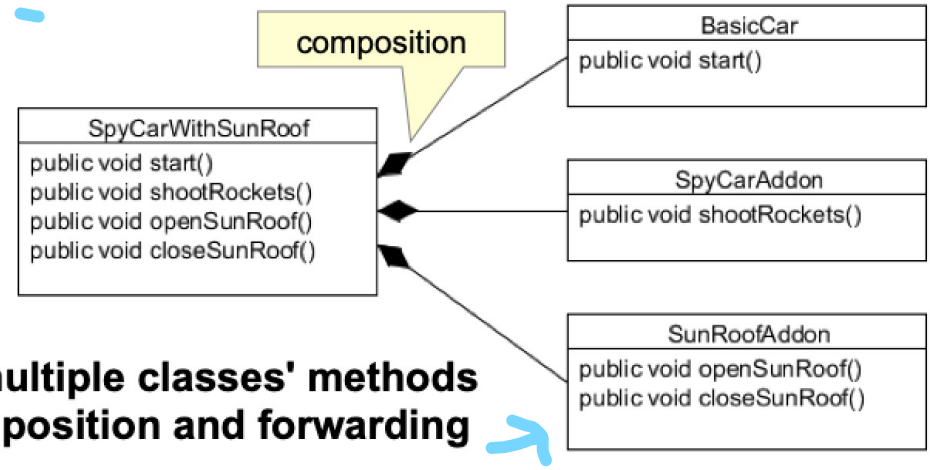
- How do you create a SpyCarWithSunRoof?



Composition

Object composition allows you to create more complex objects.
To implement composition, you:

1. Create a class with references to other classes.
2. Add same signature methods that forward to the referenced objects.



Combining multiple classes' methods through composition and forwarding

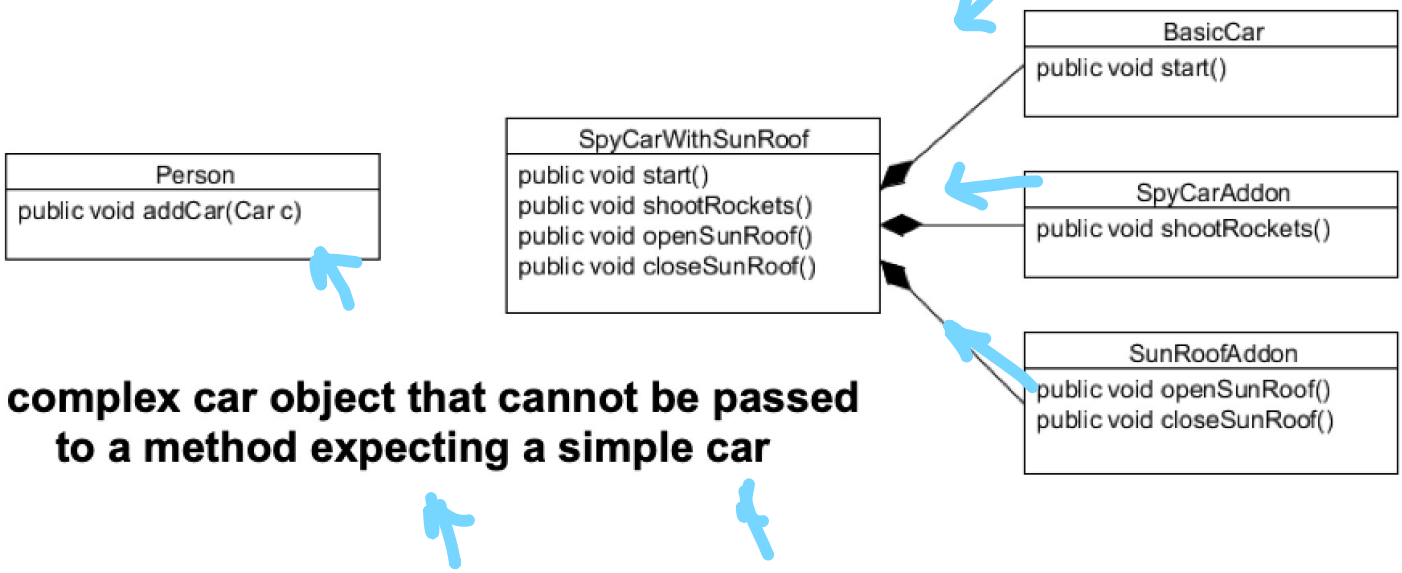
Composition Implementation

```
public class SpyCarWithSunRoof {
    private BasicCar car = new BasicCar();
    private SpyCarAddon spyAddon = new SpyCarAddon();
    private SunRoofAddon roofAddon = new SunRoofAddon();

    public void start() {
        car.start();
    }

    // other forwarded methods
}
```

Method forwarding

Polymorphism and Composition

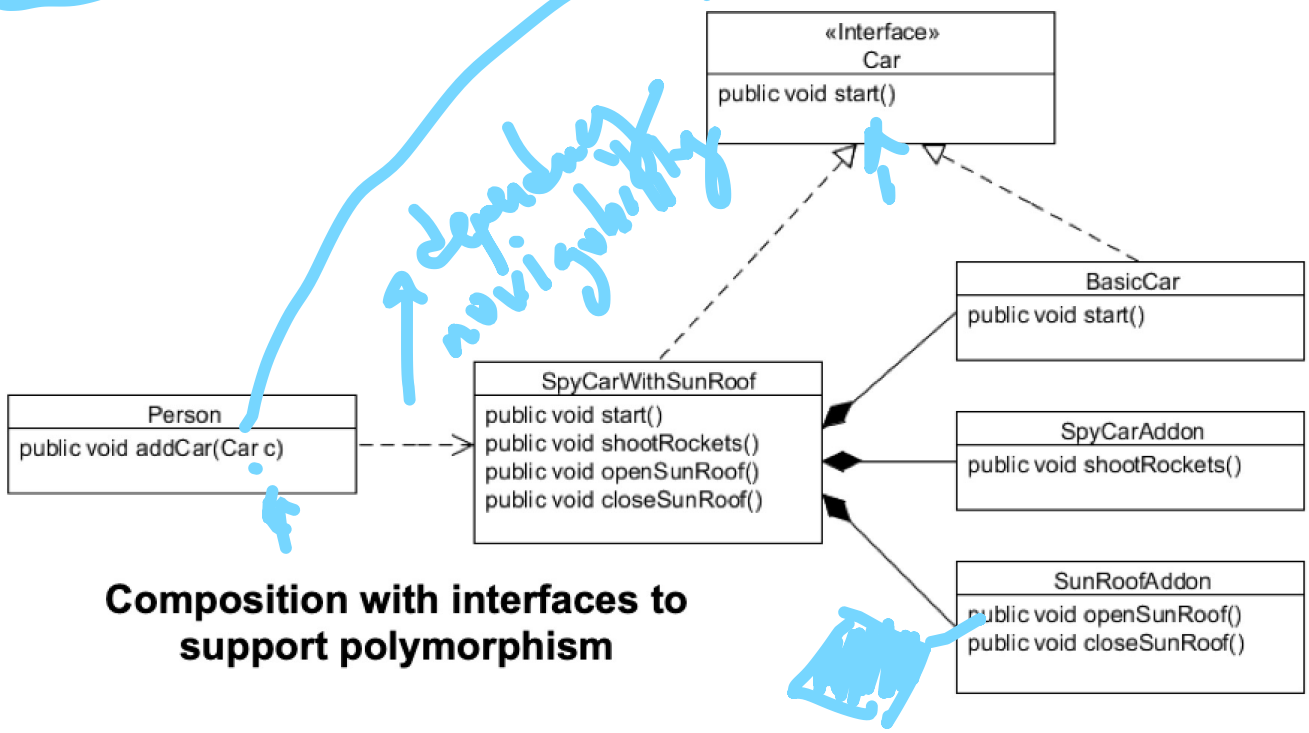
Polymorphism should enable us to pass any type of Car to the `addCar` method. Composition does not enable polymorphism unless…



A complex car object that cannot be passed to a method expecting a simple car

Polymorphism and Composition

Use interfaces for all delegate classes to support polymorphism.



Composition with interfaces to support polymorphism