Object-Oriented Programming

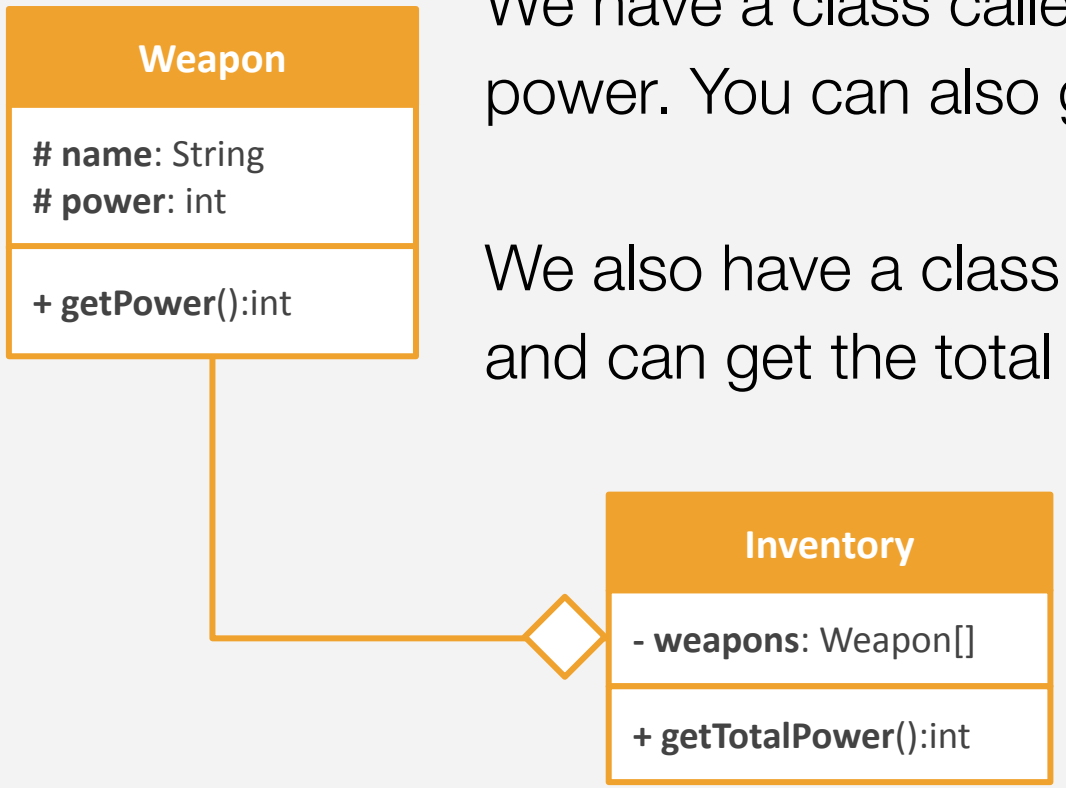
SOLID Design Principles

Slides adopted from Mr. Ryan Dimaunahan

Outline

- Motivation
- Symptoms of Bad Design
- SOLID Principles
- Exercise
- Some Announcements

Let's look at a scenario…



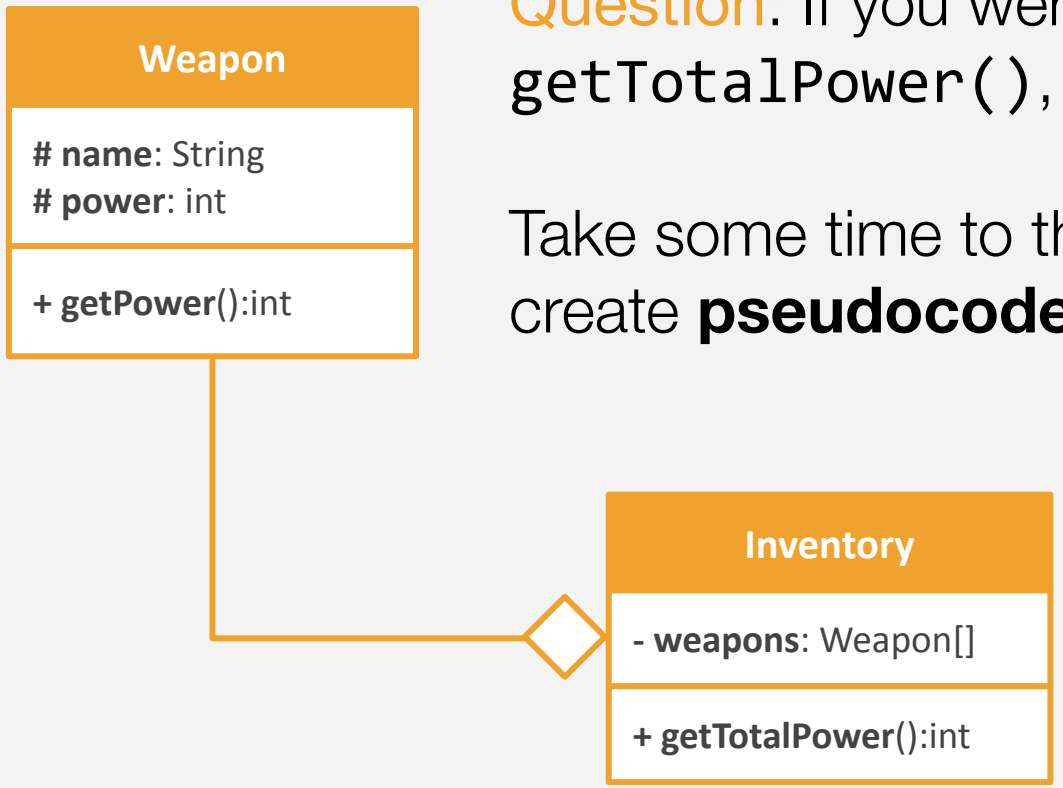
We have a class called Weapon that has an integer property called power. You can also get the power value from the class.

We also have a class called Inventory that has a list of Weapons and can get the total power based on all the stored weapons.

Question: If you were to write `getTotalPower()`, how would it look like?

Take some time to think about it and/or create **pseudocode**.

Let's look at a scenario…



Question: If you were to write `getTotalPower()`, how would it look like?

Take some time to think about it and/or create **pseudocode**.

A solution might look like…

```
01  public int getTotalPower() {
02        int sum = 0;
03        for(int i = 0; i < weapons.length; i++)
04           sum += weapons[i].getPower();
05        return sum;
06  }
```

This is a simple approach, but it gets the job done.

In a 2nd iteration of the development, an update was introduced such that there are specific subclasses of weapons, each with their own way of computing their power.

Question: How should `getTotalPower()` be modified?

Again, take some time to think about it and/or create pseudocode.

In a 2nd iteration of the development, an update was introduced such that there are specific subclasses of weapons, each with their own way of computing their power.

One approach might be to factor in the instance of the class during computation

Task: Reflect upon this design. Is it good or bad? Why?

```
01  public int getTotalPower() {
02    int sum = 0;
03    for(int i = 0; i < weapons.length; i++) {
04      Weapon weapon = weapons[i]
05      if(weapon instanceof BladedWeapon)
06        sum += weapon.getPower() + weapon.getRarity() * 10;
07      else if(weapon instanceof RangedWeapon)
08        sum += int (weapon.getPower() * weapon.getCritical());
09      else
10        sum += weapon.getPower();
11    }
12    return sum
13  }
```

Universes of Solutions

- There are an infinite number of solutions; however, not all solutions are created equal.

SPACE A:

This space represents all possible solutions, even **non-solutions** or **partial solutions**. In **CCPROG1**, your answers to problems fall in this space.

SPACE B:

This space represents a subset of solutions that are **bug-free** and **addresses all problem requirements**. This is where your solutions ideally should be after **CCPROG1** and **CCPROG2**.

SPACE C:

This space represents a subset of solutions that are **reusable, mobile, maintainable** and **extensible**.

Solutions in this space are **highly valued** by software development teams.

- This solution exists within Space B because its bug-free.
- However, the solution is difficult to maintain and hard to reuse

```
01  public int getTotalPower() {
02    int sum = 0;
03    for(int i = 0; i < weapons.length; i++) {
04      Weapon weapon = weapons[i]
05      if(weapon instanceof BladedWeapon)
06        sum += weapon.getPower() + weapon.getRarity() * 10;
07      else if(weapon instanceof RangedWeapon)
08        sum += int (weapon.getPower() * weapon.getCritical());
09      else
10        sum += weapon.getPower();
11    }
12    return sum
13  }
```

Symptoms of Bad Design

- A system is said to have bad design if it exhibits any of these 4 symptoms:
  - Rigidity (hard to change)
  - Fragility (easy to break)
  - Immobility (cannot reuse)
  - Viscosity (band-aid solutions)

Rigidity

- Occurs when several classes in a solution are too dependent on one another
  - Changes to one class causes a dependent class(es) to change as well
- When code is too rigid, a sense of resistance is developed to improving the code

Fragility

- Also a result of high dependency between classes
- While rigidity refers to resistance to change, fragility refers to code failure as a result of changes
  - “Fixes” may cause cascading negative effects to other unexpected parts of the system
- Improving fragile code becomes risky and the system becomes less reliable

Immobility

- If we can’t reuse an existing code into our new projects or modules, then it can be considered a bad design.
  - …but why?
  - It’s because your classes are so tightly dependent that it’s impossible to isolate only certain logic that you need
- Immobile solutions lead to the duplication of certain logic

Viscosity

- When there are many ways to implement a solution, a quick or temporary solution will lead to a badly designed solution
- Viscosity or the preference of convenience can lead to bad practices – which degrades the system's quality

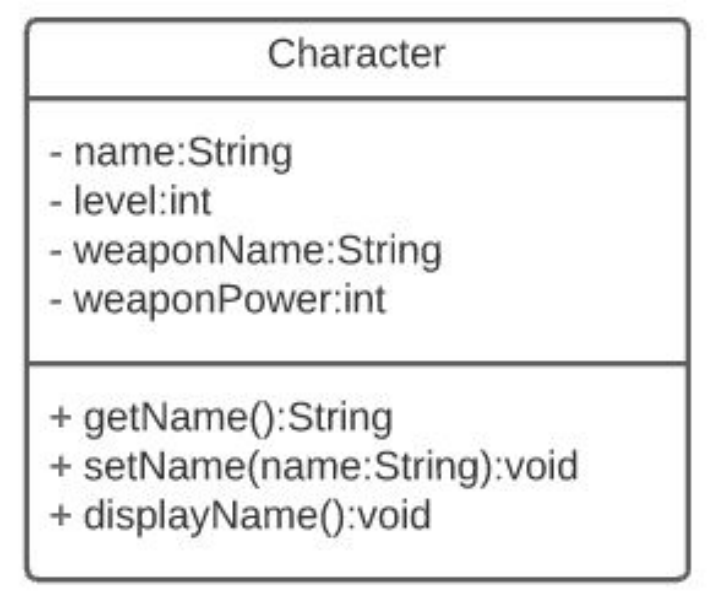
SOLID Principles

- Five principles that ensure functional code with beautiful design:
  - Single Responsibility
    - Modules have only one reason to change
  - Open-Close
    - Modules are open for extension and closed for modification
  - Liskov's Substitution
    - Subclasses must fully substitute for superclasses
  - Interface Segregation
    - Subclasses should not inherit methods they do not use
  - Dependency Inversion
    - Modules should depend on abstractions

Single Responsibility

- A class should have only one reason to change
  - If we have two reasons to change a class, then we split it into two classes
- This principle also extends to our methods
  - Each method must only accomplish one task, and each class must only represent one entity
- Idea: Recognize responsibility and delegate accordingly

Single Responsibility

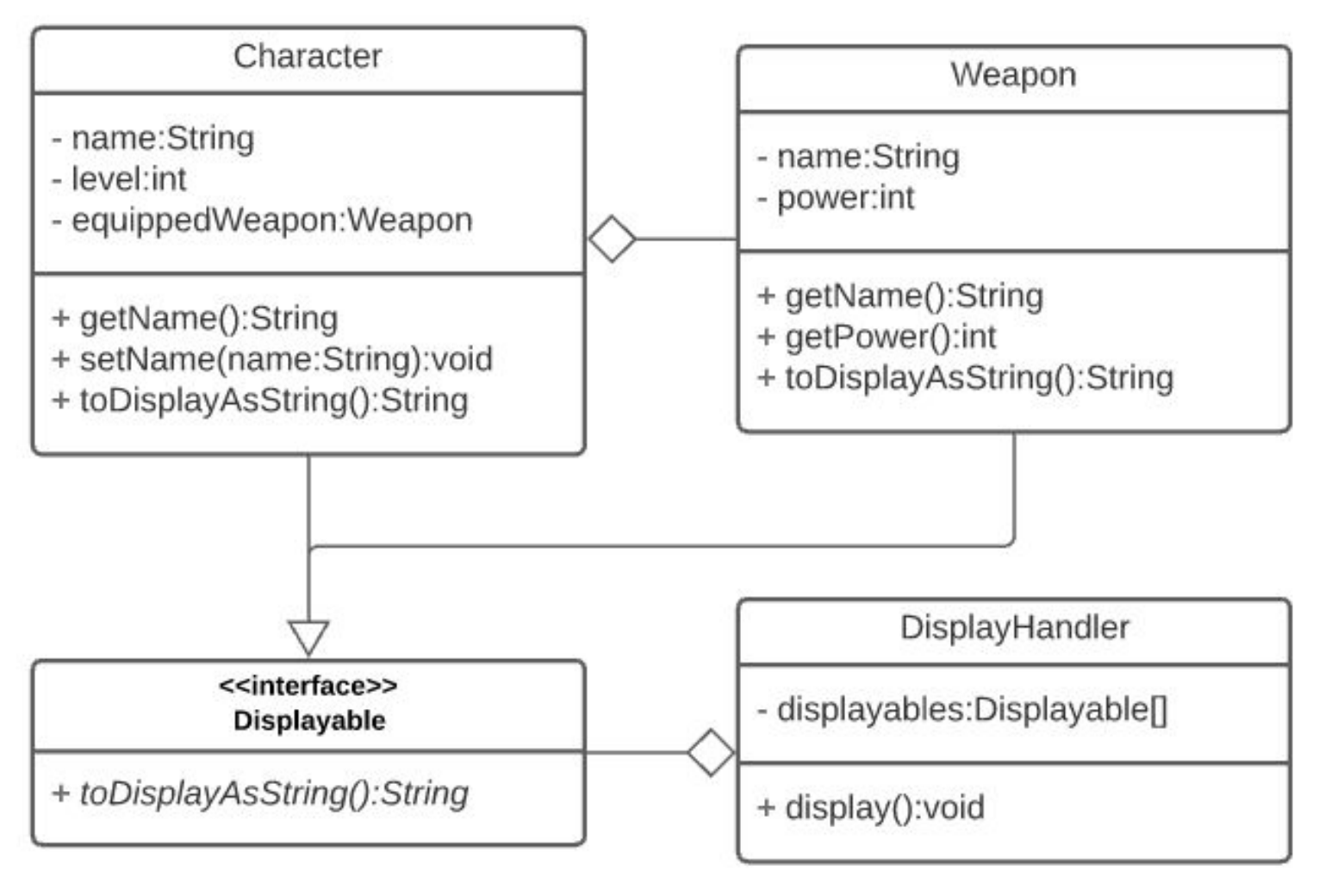


This class design is tracking too much data, and is performing too many tasks...

Reasons to change this class:

1. Changes in the implementation or design of Character objects.
2. Changes in the implementation and design of Weapon objects.
3. Changes in how screen displays are performed.

Single Responsibility



Better Design

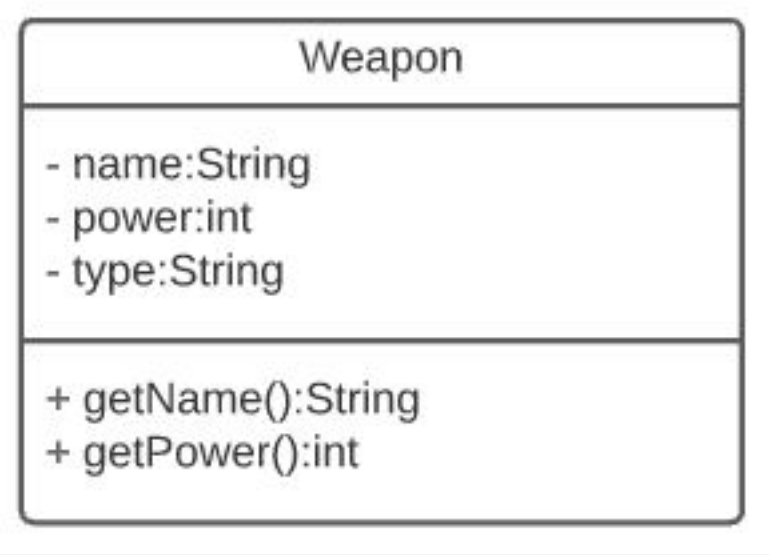
1. Each class is performing only one responsibility.
2. There is only one reason to change each class.
3. A contract/interface is used to impose to objects that can be displayed how they are going to be displayed.

Open Close Responsibility

- Classes, modules, and functions should be open for extension but closed for modifications
  - If new functionality needs to be introduced in code, I shouldn't need to open existing code because I would need to modify it!
- Solutions should provide enough extension points to shield existing code and to provide enough flexibility when adding new classes and modules

Open Close Responsibility

This design and implementation does not anticipate the inclusion of additional weapon types. If new weapon types are to be introduced, additional if-clauses have to be added to the `getPower()` method, which requires the Weapon class to be reopened...

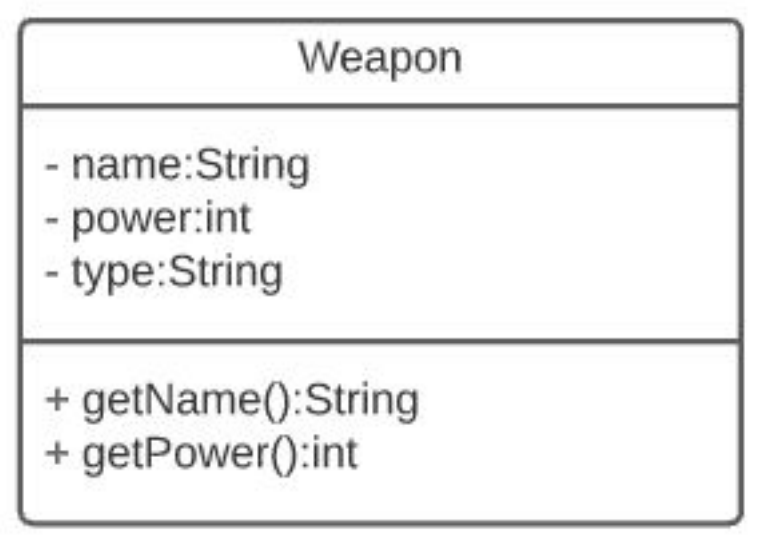



```
01 public int getPower() {
02   if (this.type.equals("Bladed"))
03     return this.power + /* some value */
04   else if (this.type.equals("Ranged"))
05     return this.power * /* some value */
06   else return this.power;
07 }
```

Good design says that classes should only be reopened if it needs fixing (say due to an undetected or unforeseen error) and not because of new functionality.

Open Close Responsibility

To anticipate the inclusion of new weapon types, we let the subclass hierarchy handle conditional behaviour through overriding!



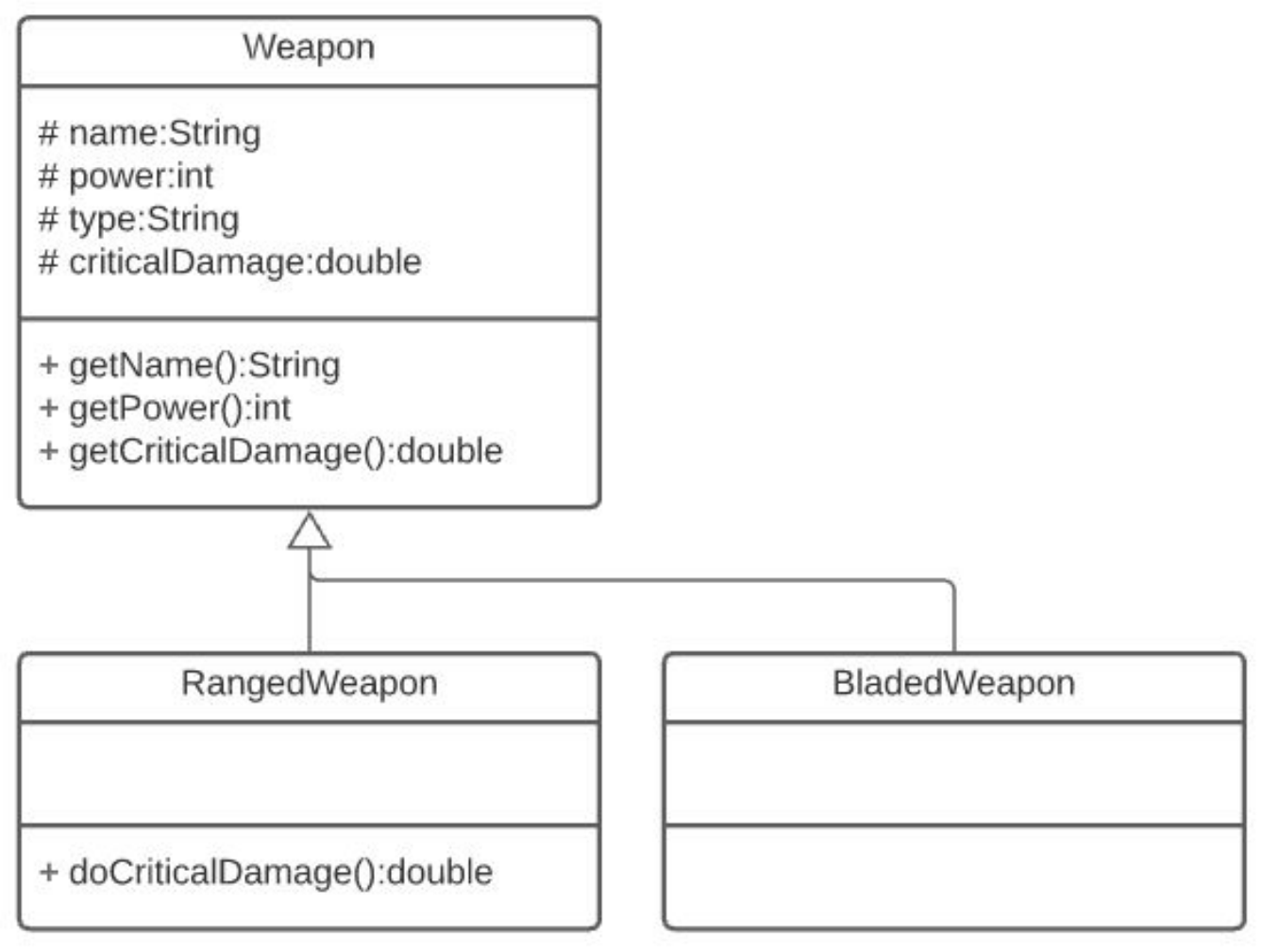

```
01 public int getPower() {
02    return this.power;
03 }
```

Should new weapon types be introduced in the future, we can extend the current Weapon class and either add new functionality or override existing functionality, replacing them with the type-specific behaviour.

Liskov's Substitution

- Derived types (subclasses) must be completely substitutable for their base types (superclasses)
  - Any subclass must do all of what the superclass can do (and then some)
  - Any subclass must have all the properties of the superclass (and then some)
- Common example given here are properties or methods that are not used by subclasses
  - Future developers may be confused as to why the method or properties exist in the first place

Liskov's Substitution

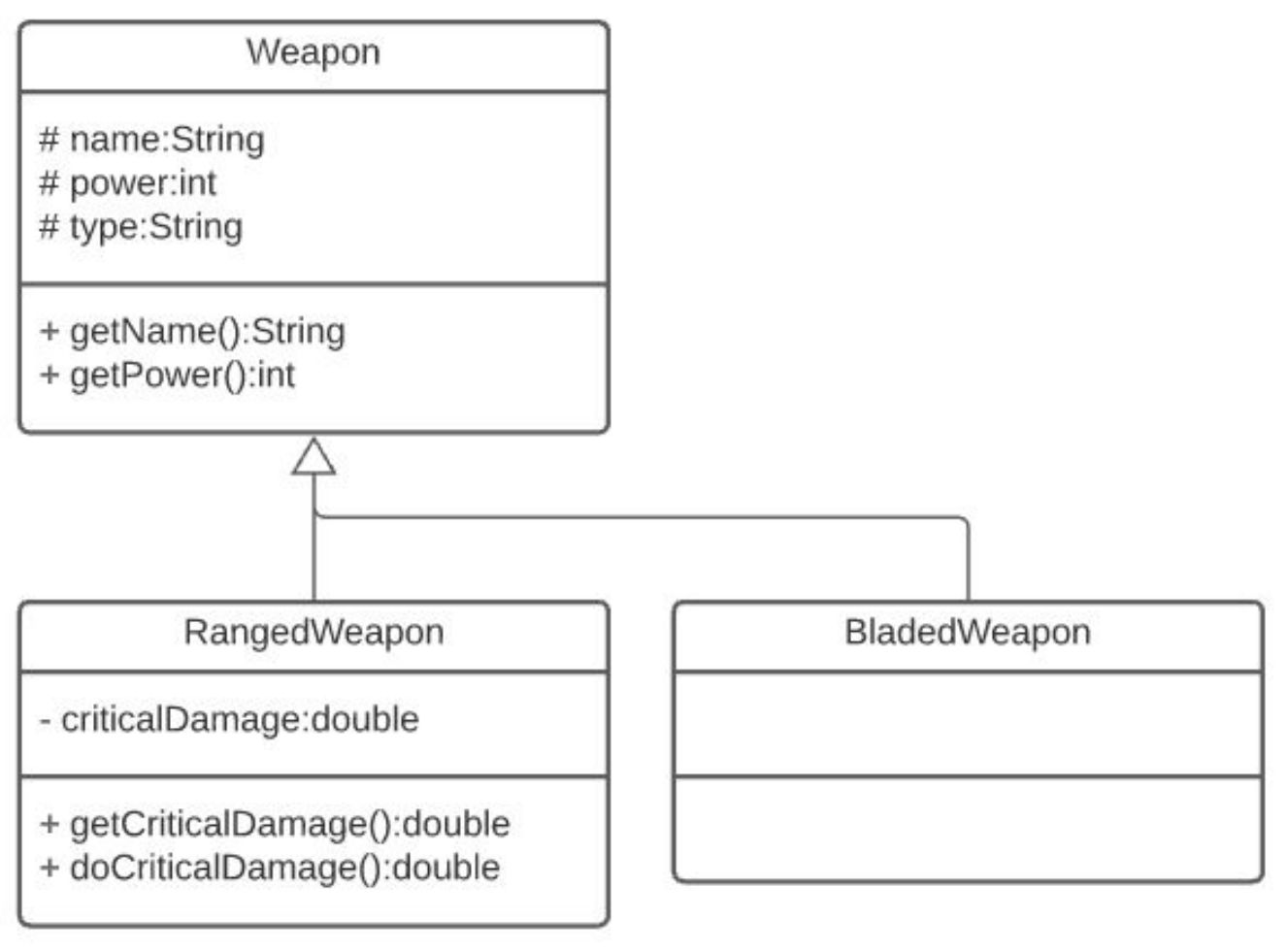


The Weapon superclass says that all Weapon types must be able to deal critical damage, as indicated by the critical damage property

This means that not all subclasses of the Weapon class are proper subclasses of Weapon.

BladedWeapon does not perform one of the behaviours of the superclass.

Liskov's Substitution

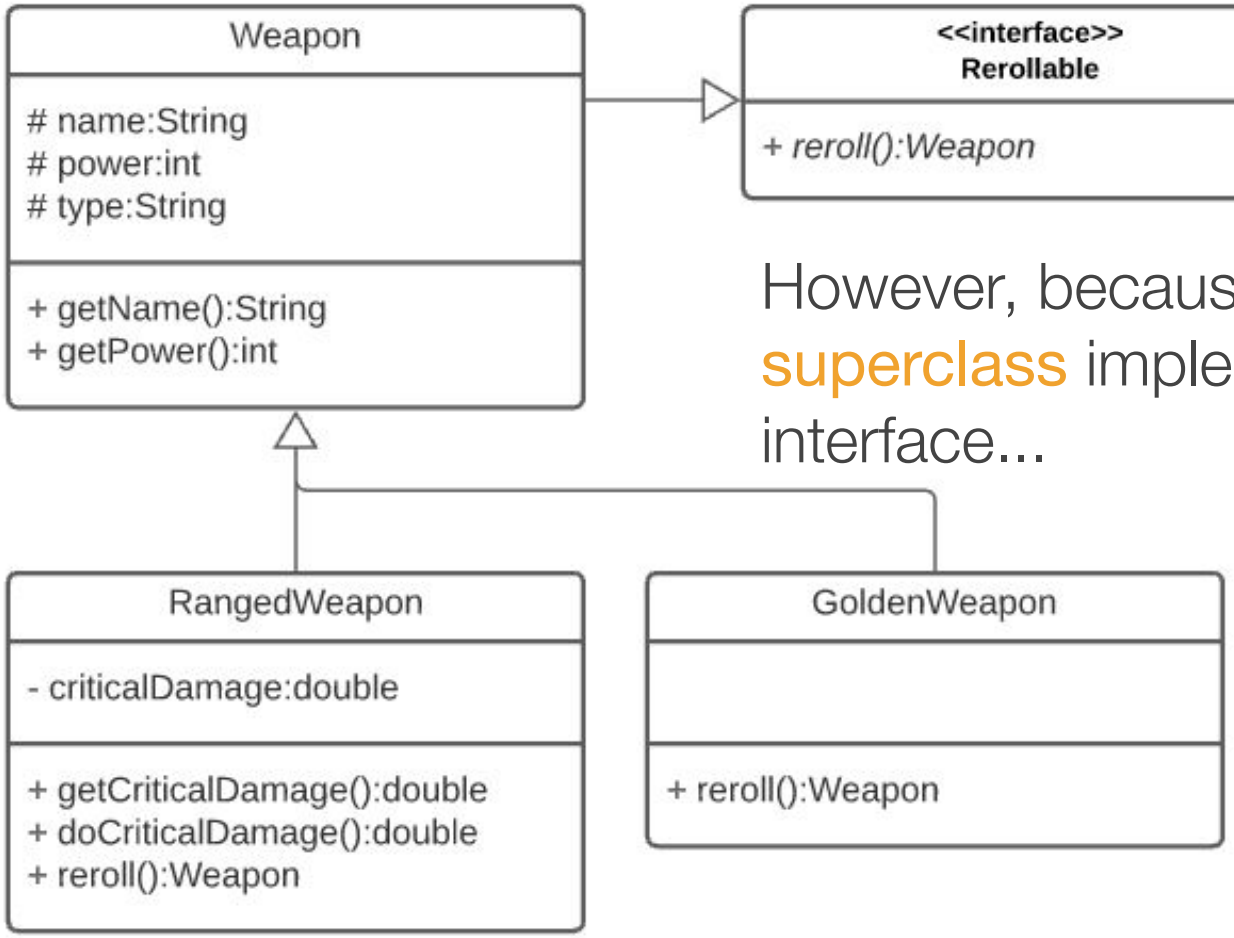


Pull down the type-specific behaviour and property to the subclass. Now both subclasses (RangedWeapon and BladedWeapon) perform all of what a Weapon does (and then some)!

Interface Segregation

- Clients (subclasses) should not be forced to depend upon interfaces (inherited abstract methods) that they don't use.
  - Subclass should not be forced to write an implementation for abstract methods that do not make sense in the context of the class
- Abstract classes and Interfaces must be designed such that all their clients would inherit meaningful abstract methods

Interface Segregation

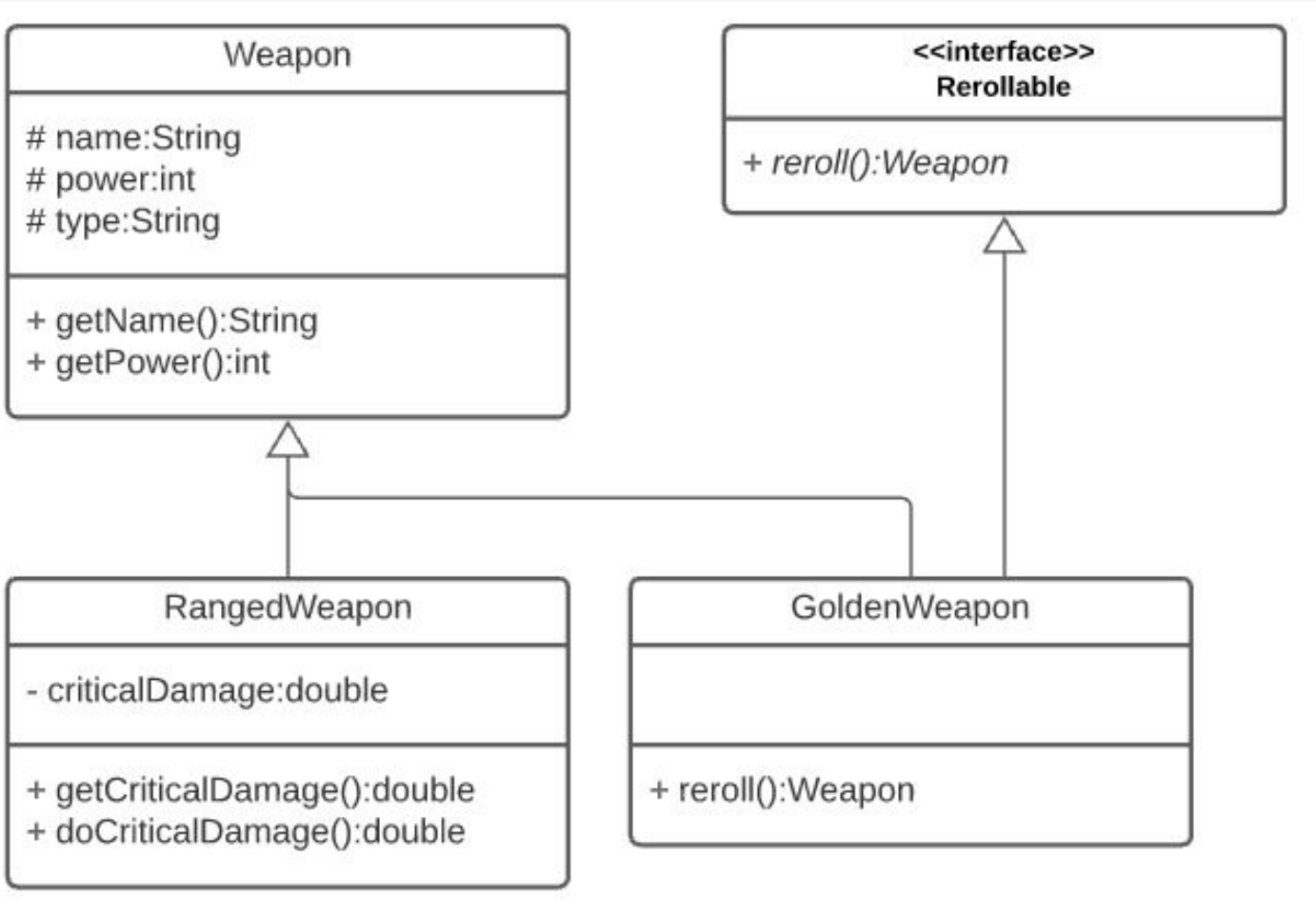


In this scenario, only GoldenWeapons should be rerollable...

However, because the Weapon superclass implements the Rerollable interface...

...all weapons are expected to implement the reroll() method, including weapons that are not GoldenWeapons.

Interface Segregation

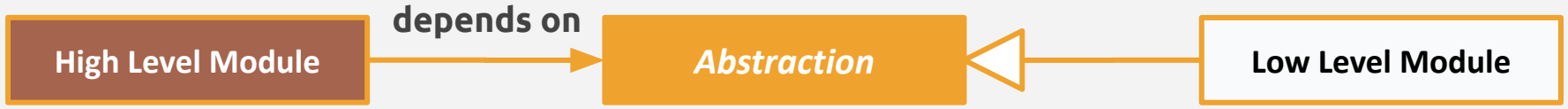


The simple fix is to have only GoldenWeapon implement the Rerollable interface. This way, it is the only class required to implement the abstract reroll() method!

This happens more frequently than you think, so be careful when designing abstractions (i.e. abstract classes and interfaces) so you do not force subclasses to implement unrelated abstract methods.

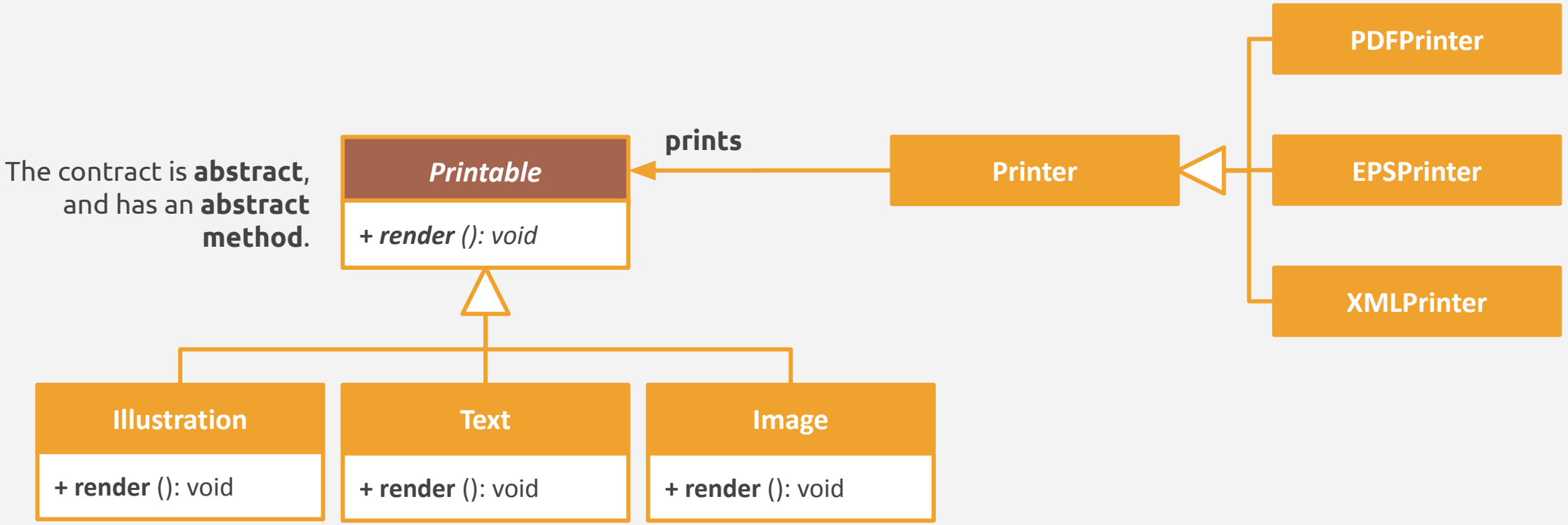
Dependency Inversion

- High-level modules should not depend on low-level modules.
  - Both should depend on abstractions.
- Abstractions should not depend on details
  - Details should depend on abstractions.



- Advocates the creation of well designed and well-made interfaces and abstract classes to facilitate communication between two modules

Dependency Inversion



Questions?

BONUS Exercise

- Serves as a Finals review
- By pair (Use only 1 computer)
- **Top 3** pairs will earn bonus points for graded exercises

Announcements

- No [online] class next week
  - See you on demo day *(choose a time schedule for your pair)*
- Demo instructions
  - Read [MCO2] Implementation Submission Page
    - F2F demo during class (earlier or may even exceed)
    - 3-minute pitch for the group
    - 2-minute Q&A for each members
- Finals - August 7 (pen-and-paper)
- Answer to Practice Exercises will be released on Aug 2

Keep learning…